Verification of WF-nets
Verification of WF-nets

TER VERKRIJGING VAN DE GRAAD VAN DOCTOR AAN DE
Technische Universiteit Eindhoven, op gezag van
de Rector Magnificus, prof.dr. R.A. van Santen,
voor een commissie aangewezen door het College
van Promoties in het openbaar te verdedigen op
donderdag 10 juni 2004 om 16.00 uur

DOOR

HENRICUS MARINUS WILHELMUS VERBEEK

geboren te Eindhoven
Dit proefschrift is goedgekeurd door de promotoren:

prof.dr.ir. W.M.P. van der Aalst
en
prof.dr. K.M. van Hee

Copromotor:
dr.ir. T. Basten
“Quod scripsi, scripsi”

Pontius Pilate
13 years ago, I started as a scientific engineer at the department of Mathematics and Computing Science, Eindhoven. When I left the military service, I had to find a job. So, I went to see Kees van Hee and Lou Somers to ask them whether I could mention them as references when applying for a job. Instead, Marc Voorhoeve told me that there just might be a vacant position that would suit me fine. For my master’s thesis, I had been working on ExSpect, and the scientific engineer who happened to be working on ExSpect had just left for the United States. To make a long story short, by coincidence, I became a scientific engineer. For several years, I worked on the ExSpect toolset.

In the mean time, Kees van Hee moved to Bakkenist Management Consultants (nowadays a part of Deloitte and Touche), which left the group more or less headless for some time. At that time, Wil van der Aalst was still only an assistant professor. However, as his star was rising with his Petri-net related work on workflow management, he was promoted to associate professor and he took over the control of the group. As a result, at the end of 1996, we started working on Woflan: a tool that exploits Petri-net analysis techniques to verify existing workflow process definitions.

So far, I had only been involved in developing software (ExSpect, Woflan), and had never given much thought on doing research. However, this was about to change. Wil encouraged me to take up this part of my job that had long since been forgotten (so it seemed). In 2001, I had cooperated in several publications and the time seemed perfect for another step: writing this PhD thesis. Wil probed me whether I would like to do such a thing, and we both concluded it would be a good idea. At the moment, you’re reading the end result of this decision.
From the above, it will be clear that I wouldn’t have written this thesis if it were not for the help of many people. First of all, of course, my parents. The first two years at secondary school were not easy for me, and they helped me through this phase with their support. I also want to thank the people from my secondary school for allowing me to continue at a level which seemed at that time too difficult for me. In contrast with secondary school, university went fine. Still, I want to thank some of my brother students: Eduard Diepstraten, Rik Koenders, Mark van Helvoirt, Han Toan Lim, and Math van Diepen. Thanks to you, I had a good time at university. Next, I would like to thank the members of the Information Systems group at the department of Mathematics and Computing Science group, and especially Kees van Hee, Lou Somers, Marc Voorhoeve, Wil van der Aalst, Twan Basten, Marco Langenhuizen, and Jeroen Schuijt. In 2000, I followed Wil when he moved to the Information and Technology group of the Technology Management department. I want to thank all members of this group for their support and trust in me. I am indebted to Wil van der Aalst for his encouragements and support over the years, and to Kees van Hee and Twan Basten for supervising me while writing this thesis. Last, but not least, I want to thank three persons that are very dear to me. Marielle, Marijn, Vera, this one’s for you:

Cliché, Fish

A concluding note on the jester figure that appears on the cover. Some of the readers might recognize this figure from Marillion’s Real to Reel album. These readers might also know that this figure is an abstraction from the jester figure as it appears on their Script for a Jester’s Tear album. As such, this jester figure on the cover is an abstraction from an abstraction. A workflow net is also an abstraction from an abstraction, as the workflow process model the workflow net is based on is already an abstraction of the real-world process. Therefore, I linked the workflow net on the cover to the jester.
# CONTENTS

## 1 Introduction
1.1 Example ....................................................... 2  
1.2 Woflan ....................................................... 7  
1.3 Road map ..................................................... 9

## 2 Workflow management
2.1 Workflow ..................................................... 13  
2.2 Abstractions .................................................. 17  
   2.2.1 Perspectives ........................................... 17  
   2.2.2 Abstraction from resources ............................ 20  
   2.2.3 Abstraction from data and triggers .................. 21  
   2.2.4 Abstraction from task content and operations ...... 22  
2.3 Verification issues ......................................... 22  
   2.3.1 Soundness ............................................. 23  
   2.3.2 Inheritance of behavior ............................... 24  
2.4 Approach ..................................................... 25  
2.5 Related work ............................................... 28  
   2.5.1 Workflow management systems ....................... 28  
   2.5.2 Workflow process definitions ....................... 29  
   2.5.3 Verification of workflow process definitions ...... 30  
   2.5.4 Petri nets ............................................. 32  
   2.5.5 WF-nets .............................................. 32  
   2.5.6 Base of this thesis .................................. 33  
2.6 Conclusions ............................................... 34

## 3 Nets and properties
3.1 Nets ......................................................... 37  
   3.1.1 Structure .............................................. 38  
   3.1.2 Net state .............................................. 42
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1.3</td>
<td>Firing rule</td>
<td>44</td>
</tr>
<tr>
<td>3.2</td>
<td>Running example</td>
<td>45</td>
</tr>
<tr>
<td>3.3</td>
<td>WF-nets</td>
<td>46</td>
</tr>
<tr>
<td>3.4</td>
<td>Net properties</td>
<td>49</td>
</tr>
<tr>
<td>3.4.1</td>
<td>Structural properties</td>
<td>49</td>
</tr>
<tr>
<td>3.4.2</td>
<td>Behavioral properties</td>
<td>56</td>
</tr>
<tr>
<td>3.5</td>
<td>System equivalence</td>
<td>67</td>
</tr>
<tr>
<td>3.6</td>
<td>Conclusions</td>
<td>72</td>
</tr>
<tr>
<td>4</td>
<td>Deciding soundness</td>
<td>73</td>
</tr>
<tr>
<td>4.1</td>
<td>Structural properties related to soundness</td>
<td>73</td>
</tr>
<tr>
<td>4.2</td>
<td>Liveness and boundedness vs. soundness</td>
<td>79</td>
</tr>
<tr>
<td>4.2.1</td>
<td>Proper completion</td>
<td>79</td>
</tr>
<tr>
<td>4.2.2</td>
<td>Option to complete</td>
<td>81</td>
</tr>
<tr>
<td>4.2.3</td>
<td>No dead tasks</td>
<td>82</td>
</tr>
<tr>
<td>4.3</td>
<td>Behavioral error sequences</td>
<td>83</td>
</tr>
<tr>
<td>4.3.1</td>
<td>Non-live sequences</td>
<td>84</td>
</tr>
<tr>
<td>4.3.2</td>
<td>Unbounded sequences</td>
<td>88</td>
</tr>
<tr>
<td>4.4</td>
<td>Reduction techniques</td>
<td>92</td>
</tr>
<tr>
<td>4.4.1</td>
<td>Liveness and boundedness preserving reduction rules</td>
<td>93</td>
</tr>
<tr>
<td>4.4.2</td>
<td>Reduction rules for free-choice nets</td>
<td>94</td>
</tr>
<tr>
<td>4.4.3</td>
<td>Restricted coverability graph</td>
<td>95</td>
</tr>
<tr>
<td>4.5</td>
<td>Diagnosis process</td>
<td>96</td>
</tr>
<tr>
<td>4.5.1</td>
<td>Step 1: Start of diagnosis</td>
<td>98</td>
</tr>
<tr>
<td>4.5.2</td>
<td>Step 2: Workflow process definition?</td>
<td>98</td>
</tr>
<tr>
<td>4.5.3</td>
<td>Step 3: Thread of control cover?</td>
<td>99</td>
</tr>
<tr>
<td>4.5.4</td>
<td>Step 4: Confusions and mismatches?</td>
<td>100</td>
</tr>
<tr>
<td>4.5.5</td>
<td>Step 5: Uniform invariant cover?</td>
<td>104</td>
</tr>
<tr>
<td>4.5.6</td>
<td>Step 6: Weighted invariant cover?</td>
<td>104</td>
</tr>
<tr>
<td>4.5.7</td>
<td>Step 7: No improper conditions?</td>
<td>105</td>
</tr>
<tr>
<td>4.5.8</td>
<td>Step 8: No subtstates?</td>
<td>105</td>
</tr>
<tr>
<td>4.5.9</td>
<td>Step 9: Improper scenarios!</td>
<td>106</td>
</tr>
<tr>
<td>4.5.10</td>
<td>Step 10: No dead tasks?</td>
<td>107</td>
</tr>
<tr>
<td>4.5.11</td>
<td>Step 11: No non-live tasks?</td>
<td>108</td>
</tr>
<tr>
<td>4.5.12</td>
<td>Step 12: Non-live tasks!</td>
<td>108</td>
</tr>
<tr>
<td>4.5.13</td>
<td>Step 13: Locking scenarios!</td>
<td>108</td>
</tr>
<tr>
<td>4.5.14</td>
<td>Step 14: End of diagnosis</td>
<td>109</td>
</tr>
<tr>
<td>4.5.15</td>
<td>Example: the free-choice Revised net</td>
<td>109</td>
</tr>
<tr>
<td>4.6</td>
<td>Case study</td>
<td>110</td>
</tr>
<tr>
<td>4.7</td>
<td>Conclusions</td>
<td>115</td>
</tr>
<tr>
<td>5</td>
<td>Deciding life-cycle inheritance</td>
<td>117</td>
</tr>
<tr>
<td>5.1</td>
<td>Exhaustive search algorithm</td>
<td>117</td>
</tr>
<tr>
<td>5.2</td>
<td>Constraints</td>
<td>118</td>
</tr>
<tr>
<td>5.3</td>
<td>Successful termination</td>
<td>121</td>
</tr>
<tr>
<td>5.4</td>
<td>Matching cycles</td>
<td>125</td>
</tr>
<tr>
<td>5.4.1</td>
<td>Block constraints</td>
<td>127</td>
</tr>
</tbody>
</table>
## Contents

5.4.2 Hide constraints ........................................... 128  
5.4.3 Structural properties ....................................... 129  
5.4.4 Example ................................................... 133  
5.5 Backtracking algorithm ........................................ 134  
5.6 Case study .................................................... 137  
5.7 Conclusions ................................................... 139  

6 XRL ........................................ 141  
6.1 Motivation .................................................... 142  
6.2 An XML-based routing language ................................ 144  
6.2.1 Syntax of XRL ............................................. 144  
6.3 WF-net semantics of XRL ....................................... 149  
6.3.1 Control passing and proper completion ....................... 150  
6.3.2 Events ...................................................... 152  
6.3.3 Terminates .................................................. 153  
6.3.4 Route element ............................................. 154  
6.3.5 Terminate element ......................................... 155  
6.3.6 Task element ............................................... 156  
6.3.7 Sequence element ......................................... 157  
6.3.8 Any_sequence element ...................................... 158  
6.3.9 Choice element ............................................ 158  
6.3.10 Parallel_sync element .................................... 159  
6.3.11 Parallel_no_sync element ................................ 159  
6.3.12 Parallel_part_sync element ................................ 160  
6.3.13 Parallel_part_sync_cancel element ......................... 162  
6.3.14 Wait_all element ......................................... 162  
6.3.15 Wait_any element ......................................... 163  
6.3.16 While_do element ........................................ 163  
6.3.17 Condition element ....................................... 164  
6.3.18 Mapping onto a WF-net ................................... 166  
6.4 Verification of XRL ............................................ 167  
6.4.1 Proper completion ......................................... 168  
6.4.2 Option to complete and no dead tasks ...................... 173  
6.4.3 Reduction techniques ..................................... 175  
6.4.4 Verification procedure .................................... 179  
6.5 Conclusions ................................................... 179  

7 Mappings ........................................ 181  
7.1 MQSeries Workflow ........................................... 183  
7.1.1 Processes .................................................. 183  
7.1.2 Sample processes ......................................... 184  
7.1.3 Mapping ...................................................... 185  
7.1.4 WF-nets ...................................................... 190  
7.1.5 Soundness ................................................... 190  
7.1.6 Life-cycle inheritance ..................................... 192  
7.1.7 Summary .................................................... 192  
7.2 Staffware 2000 .................................................. 193
# Contents

7.2.1 Processes ................................................. 193
7.2.2 Sample processes ......................................... 195
7.2.3 Mapping .................................................. 196
7.2.4 WF-net ................................................... 199
7.2.5 Soundness ................................................ 202
7.2.6 Semantical issues ........................................ 203
7.2.7 Summary .................................................. 205

7.3 COSA Workflow ............................................. 206
7.3.1 Processes ................................................. 206
7.3.2 Sample processes ......................................... 207
7.3.3 Mapping .................................................. 208
7.3.4 WF-net ................................................... 210
7.3.5 Soundness ................................................ 211
7.3.6 Semantical issues ........................................ 212
7.3.7 Summary .................................................. 212

7.4 Protos ....................................................... 212
7.4.1 Processes ................................................. 213
7.4.2 Mapping .................................................. 213

7.5 Conclusions ................................................. 215

8 Woflan ......................................................... 217
8.1 Architecture .................................................. 217
8.1.1 Static library grcps.lib .................................. 218
8.1.2 Static library woflib.lib .................................. 220
8.1.3 Static library wofdll.lib .................................. 221
8.1.4 Dynamic library wofdll.dll ................................ 222
8.1.5 Static library co2tpn.lib .................................. 223
8.1.6 Static library xfr2tpn.lib .................................. 224
8.1.7 Static library ...2tpn.lib .................................. 224
8.1.8 Executable wofapp.exe ...................................... 224
8.2 Windows application ......................................... 224
8.2.1 Diagnosis process ......................................... 225
8.2.2 Life-cycle inheritance ...................................... 230
8.3 Web service application .................................... 232
8.4 Ongoing work ............................................... 234
8.5 Conclusions ............................................... 235

9 Case studies .................................................. 237
9.1 Staffware case ............................................... 237
9.1.1 Diagnosing and correcting the process definition ............ 238
9.1.2 Summary ............................................... 242
9.2 Procurement case ............................................ 243
9.2.1 Receive Materials Against P.O ................................ 244
9.2.2 Receive Goods ............................................. 247
9.2.3 Procurement .............................................. 249
9.2.4 Summary ............................................... 250
9.3 Transit case ................................................. 250

---

xii
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>9.3.1 Performance results</td>
<td>251</td>
</tr>
<tr>
<td>9.3.2 Diagnosing and correcting the processes</td>
<td>252</td>
</tr>
<tr>
<td>9.3.3 Performance results on the corrected processes</td>
<td>261</td>
</tr>
<tr>
<td>9.3.4 Summary</td>
<td>263</td>
</tr>
<tr>
<td>9.4 Conclusions</td>
<td>264</td>
</tr>
<tr>
<td>10 Concluding Remarks</td>
<td>265</td>
</tr>
<tr>
<td>10.1 Contributions</td>
<td>265</td>
</tr>
<tr>
<td>10.1.1 Diagnosis process and behavioral error sequences</td>
<td>265</td>
</tr>
<tr>
<td>10.1.2 Backtracking algorithm</td>
<td>266</td>
</tr>
<tr>
<td>10.1.3 Mappings</td>
<td>267</td>
</tr>
<tr>
<td>10.1.4 Woflan</td>
<td>267</td>
</tr>
<tr>
<td>10.1.5 Case studies</td>
<td>267</td>
</tr>
<tr>
<td>10.2 Limitations and future work</td>
<td>268</td>
</tr>
<tr>
<td>10.2.1 Control-flow perspective</td>
<td>268</td>
</tr>
<tr>
<td>10.2.2 WF-nets</td>
<td>269</td>
</tr>
<tr>
<td>10.2.3 Soundness</td>
<td>271</td>
</tr>
<tr>
<td>10.2.4 Inheritance</td>
<td>273</td>
</tr>
<tr>
<td>10.2.5 Woflan</td>
<td>275</td>
</tr>
<tr>
<td>A WoflanInfo</td>
<td>277</td>
</tr>
<tr>
<td>A.1 Net</td>
<td>278</td>
</tr>
<tr>
<td>A.2 WF-net</td>
<td>278</td>
</tr>
<tr>
<td>A.3 S-components</td>
<td>279</td>
</tr>
<tr>
<td>A.4 Place Invariants</td>
<td>279</td>
</tr>
<tr>
<td>A.5 Transition invariants</td>
<td>280</td>
</tr>
<tr>
<td>A.6 Free choice</td>
<td>281</td>
</tr>
<tr>
<td>A.7 Handles</td>
<td>281</td>
</tr>
<tr>
<td>A.8 Unboundedness</td>
<td>281</td>
</tr>
<tr>
<td>A.9 Liveness</td>
<td>282</td>
</tr>
<tr>
<td>A.10 Inheritance</td>
<td>282</td>
</tr>
<tr>
<td>A.11 Minimal coverability graph</td>
<td>283</td>
</tr>
<tr>
<td>A.12 Miscellaneous</td>
<td>283</td>
</tr>
<tr>
<td>B The TPN format</td>
<td>285</td>
</tr>
<tr>
<td>B.1 Extended BNF</td>
<td>285</td>
</tr>
<tr>
<td>B.2 Example</td>
<td>286</td>
</tr>
<tr>
<td>C Travel agency at Somewhere University</td>
<td>289</td>
</tr>
<tr>
<td>D XRL DTD</td>
<td>293</td>
</tr>
<tr>
<td>Bibliography</td>
<td>297</td>
</tr>
<tr>
<td>Index</td>
<td>307</td>
</tr>
<tr>
<td>Summary</td>
<td>317</td>
</tr>
<tr>
<td>Samenvatting</td>
<td>319</td>
</tr>
<tr>
<td>Curriculum vitae</td>
<td>321</td>
</tr>
</tbody>
</table>

xiii
Workflow management systems take care of the automated support and coordination of business processes to reduce costs and flow times and to increase quality of service and productivity [HL91, JB96, Kou95, Law97, Fis01, Sch96]. A critical challenge for workflow management systems is their ability to respond effectively to changes in business processes [ADE98, ADO00, CCPP98, KDB98, RD98, WR96]. Changes may range from simple modifications of a workflow process such as adding a task to a complete restructuring of the workflow process to improve efficiency. Changes may also involve the creation of new processes.

Today’s workflow management systems are ill suited to dealing with frequent changes, because there are hardly any checks to assure some minimal level of correctness. Creating or modifying a complex process that combines parallel and conditional routing is an activity subject to errors. Even a simple change such as adding a task can cause a deadlock or livelock.

A deadlock occurs if at some unexpected point in the workflow process it is no longer possible to make any progress for a case (workflow instance) that is being handled. Note that the expected termination of progress is something desirable, because it corresponds to the successful completion of a case. A livelock occurs if it is possible to make continuous progress for a certain case, however, without progressing towards successful completion and without ending in a deadlock (that is, an endless loop).

Contemporary workflow management systems do not support advanced techniques to verify the correctness of workflow process def-
Introduction

These systems typically restrict themselves to a number of (trivial) syntactical checks. Therefore, serious errors such as deadlocks and livelocks may remain undetected. This means that an erroneous workflow may go into production, thus causing problems for the organization. An erroneous workflow may lead to extra work, legal problems, dissatisfied customers, managerial problems, and depressed employees. Therefore, it is important to verify the correctness of a workflow process definition before it becomes operational. The role of verification becomes even more important as many enterprises are making Total Quality Management (TQM) one of their focal points. For example, ISO 9000 certification and compliance forces companies to document business processes and to meet self-imposed quality goals [IC96]. Clearly, rigorous verification of workflow process definitions can be used as a tool to ensure certain levels of quality.

1.1 Example

We use a requisition process based on the one found in [Ora03] as an example to show that errors can be introduced easily using contemporary workflow-system-related modeling tools. Through this requisition process, employees of some company are able to purchase items they need for doing their jobs. After submitting a request for requisition, the manager of the requestor is selected to approve the requisition. If the requestor has no manager, then the requisition is rejected immediately. Otherwise, two tasks are started in parallel: we record the fact that the approval of the requisition is forwarded to the selected manager, and we notify the requestor of this forward. After both tasks have been completed, we notify the selected manager that there is a requisition for him to approve. If the manager does not react in a timely manner, s/he is notified again, until s/he either approves or rejects the requisition. If s/he approves the requisition, we verify whether the manager’s spending limit is sufficient for the requisition. If so, the approval of the requisition is recorded and the requestor is notified. If not, the requisition is forwarded for approval to the next higher manager.

Figure 1.1 shows an attempt to model this process using the Network Editor of the workflow management system COSA [SL98, Baa99]. However, when enacting this COSA model by the COSA workflow management system, some requisition requests might not be able to complete, that is, they might not be able to reach the end of the process. For example the following scenario can occur according to this model:

1. A request for a requisition is submitted (Routing construct StartOf Requisition).
2. The requestor has no manager; therefore, the false branch is activated (task Select approver).

At this point, the requisition request gets stuck: The task Notify requestor of rejection needs both its inputs to be available, but the input Rejected is not available. A second scenario is the following:

1. A request for a requisition is submitted (Routing construct StartOf Requisition).

2. The requestor has a manager; therefore, the true branch is activated (task Select approver).

3. Two parallel branches are started (Routing construct AND split).

4. The forward of the requisition request to the manager is recorded (Task Record requisition forward).

5. The requestor is notified about this forward (Task Notify requestor of forward).

6. The two parallel branches are synchronized (Routing construct AND join).

7. The manager of the requestor decides to reject the requisition; therefore, the Reject branch is activated (Task Notify approver).

At this point, the requisition gets stuck too: The input No manager is not available.

The direct cause of this problem is the task Notify requestor of rejection, which needs both its inputs (Rejected and No manager) to be available, although at most one of them can be available: If no
approver could be found (that is, if branch false is activated), then no approver could reject the requisition (that is, branch reject cannot get activated). Thus, a deadlock occurs if the requisition is rejected. Note that because of this deadlock, the task Notify requestor of rejection will never be executed, that is, it is dead. As a result, any case corresponding to a rejected requisition will never be completed and these 'rejected' cases might clog the entire workflow system. Figure 1.2 shows how we can correct this error in COSA: We remove the condition No manager after we routed its input arc to the condition Rejected. As a result, the Notify requestor of rejection task can start as soon as condition Rejected is available, and this condition will be available if either no appropriate approver can be found (false) or some manager rejects the requisition (Reject).

Figure 1.3 shows another attempt to model the requisition process, this time using the Graphical Workflow Definer (GWD) [Sta97, Sta99] of the Staffware workflow management system. However, when this model is enacted by the Staffware workflow system, requisition requests may exhibit improper behavior. For example, the following scenario can occur:

1. An approver is successfully selected (task Select approver), which enables two tasks: Notify requestor of fwd and Record requisition forward.
2. The requestor is notified on the forward (task Notify requestor of fwd).
3. The forward is recorded (task Record requisition fwd).
4. The manager takes too long to approve or reject the requisition, thus the deadline expires. As a result, both the task Notify approver and the task Remind approver are now available.

When a deadline is associated to a task in Staffware, then the process designer can set the deadline to either withdraw that task or to keep that task enabled if the deadline expires. In the case of task Notify approver, the process designer set the deadline to keep it enabled.

5. The approver approves the requisition (task Notify approver).

6. The approver receives a reminder to either approve or reject the requisition (task Remind approver), which enables the task Notify approver.

7. The authority of the manager is verified successfully (task Verify Authority).

8. The requestor is notified of this approval (task Notify requestor of app).

9. The approver reacts to the reminder (see step 6: task Notify approver is still enabled) by rejecting the requisition (task Notify approver).

10. The requestor is notified of this rejection (task Notify requestor of rej).

Thus, the requestor of this requisition receives both a notification of approval and a notification of rejection, which is clearly an error. The direct cause of this error is the task Notify approver with its associated deadline. Figure 1.4 shows the configuration dialog for this deadline in the Staffware process definition. This dialog clearly shows that the task Notify approver is not withdrawn if this deadline expires. As a result, for each reminder send to a manager, that manager might have to approve the requisition request an additional time.

---

**FIGURE 1.3. A Staffware model of a Requisition process.**
Another type of errors is related to, for example, management views on workflow process definitions, where management views are typically obtained by abstracting from unwanted details. Figure 1.5 shows an attempt to model a management view on the requisition process using the business process reengineering (BPR) tool Protos [Pal97, Wav02]. This management view focuses on the requestor, and leaves all other details out. According to this management view, after the requestor has received any number of forward notifications, s/he receives either a notification of approval or a notification of rejection.

When compared to the requisition process as shown in Figure 1.2, this management view seems to be in order. However, the attentive reader may have noticed that the management view as shown in Figure 1.5 does not exactly capture the requestor’s view on the requisition process as shown in Figure 1.2: In Figure 1.5, the requestor can receive a notification of approval without having received a notification of forward, while in Figure 1.2 this is not possible. Therefore, this management view does not exactly capture the requestor’s view on the requisition process, and could thus lead to false assumptions on this process.
As the example shows, errors made are easily overlooked in a workflow process definition. Therefore, we need to verify a workflow process definition thoroughly before it is taken into production, that is, before it is enacted by some workflow system. Furthermore, the example shows that the errors made in abstractions of a workflow process definition are easily overlooked as well. Thus, we also need to be able to thoroughly check whether some abstraction matches its workflow process definition. For these reasons, we started developing the verification tool Woflan.

1.2 Woflan

This thesis presents both theoretical and practical results that have been used to develop the verification tool Woflan (workflow analyzer). Woflan analyzes workflow process definitions downloaded from commercial workflow products, using state-of-the-art Petri-net-based analysis techniques. The development of Woflan started at the end of 1996. The goal was to build a verification tool specifically designed for workflow analysis. Right from the start, there have been three important requirements for Woflan:

1. Woflan should be product independent, that is, it should be possible to analyze process definitions designed with various workflow products of different vendors.
2. Woflan should be able to handle complex workflow definitions with up to hundreds of tasks.
3. Woflan should give to-the-point diagnostic information for repairing detected errors.

At the moment, there is still no consensus among vendors of workflow management systems on how to model a workflow process definition, although the vast majority of them uses diagramming techniques. Thus, to be product independent, we need to be able to map these different diagramming techniques onto some format that we are able to use for our verification and diagnosis purposes. Furthermore, because we need to be able to handle complex workflow process definitions, efficient analysis techniques are needed. For these reasons, we decided to base Woflan on Petri nets. Petri nets are a universal modeling language with a solid mathematical foundation. Yet, Petri nets are close to the diagramming techniques used in today’s workflow management systems. The efficient analysis techniques developed for Petri nets allow for the analysis of complex workflow process definitions. Finally, the graphical representation of Petri nets and the available analysis techniques are particularly useful for generating meaningful diagnostic information.
Since the release of version 1.0 of the tool in 1997, we have been continuously improving Woflan. Both new theoretical results and practical experiences stimulated several enhancements. Pivotal to guarantee some minimal level of correctness when creating or modifying workflow process definitions, and, hence, pivotal to Woflan, are the notions of soundness of a workflow process definition [Aal97, Aal98a, Aal00c] and four notions of inheritance between sound workflow process definitions [Bas98, BA01, AB02]. The soundness notion expresses minimal requirements any workflow process definition should satisfy, and it allows us to detect the errors in the requisition processes as shown in Figure 1.1 and Figure 1.3. Informally, a workflow process definition is sound if it satisfies the following requirements.

**Option to complete.** It should always be possible to complete a case (a running instance, for example, a requisition request) that is handled according to the process. This condition guarantees the absence of deadlocks and livelocks. The COSA example shown in Figure 1.1 clearly violates this requirement, as it blocks for every rejected requisition request.

**Proper completion.** It should not be possible that the workflow process definition signals completion of a case while there is still work in progress for that case. The Staffware example shown in Figure 1.3 clearly violates this requirement, as after the requestor has received a notification of approval and his request has been recorded (which signals completion), s/he receives a notification of rejection and the request is recorded for a second time.

**No dead tasks.** For every task, there should be an execution of the workflow process definition that executes it. This restriction means that every task has a meaningful role in the workflow process. The COSA example shown in Figure 1.1 clearly violates this requirement, as the task Notify requestor of rejection can never be executed.

The four inheritance relations all assume that one of the workflow process definitions is an extension of the other, that is, the set of tasks in the second definition is a subset (but not necessarily a strict subset) of the first definition. The tasks that are only present in the first process are called extending tasks. For example, the requisition process as shown in Figure 1.2 is an extension of the requestor’s view as shown in Figure 1.5, with Select approver, Record requisition forward, Notify approver, Remind approver, Verify authority, and Record result as extending tasks. The four inheritance relations are as follows:
Protocol inheritance. The second process definition exhibits behavior equivalent to the first process definition when all extending tasks are blocked (they are not allowed to happen.)

Projection inheritance. The second process definition exhibits behavior similar to the first process definition when all extending tasks are hidden (they are allowed to happen, but they cannot be observed.)

Protocol/projection inheritance. The second process definition exhibits behavior similar to the first process definition when all extending tasks are blocked and also when all extending tasks are hidden.

Life-cycle inheritance. The second process definition exhibits behavior similar to the first process definition when some extending tasks are blocked and the remaining extending tasks are hidden.

These relations can be used, for instance, to determine that the requestor’s view as shown in Figure 1.5 does not match the requisition process as shown in Figure 1.2. Likewise, we can use these relations when we have to determine whether a process definition extends a second process definition in such a way that cases can be migrated successfully from the second to the first. This is especially important when we have to modify a complex workflow process with a lot of cases.

For this thesis, we are mainly interested in the life-cycle inheritance relation. For protocol inheritance and projection inheritance, we only have to block or hide all extending tasks and check whether the resulting definitions behave in a similar way, which can be done in an efficient way using existing algorithms. For protocol/projection inheritance we have to check for similar behavior at most twice, which is acceptable. However, for life-cycle inheritance, we possibly have to check all combinations of blocking and hiding the extending tasks, which leads to an additional exponential factor. Because of its additional complexity, the life-cycle inheritance relation is of special interest to this thesis.

1.3 Road map

The remainder of this thesis is organized as follows.

Chapter 2 introduces workflow management and signals that the current workflow products lack decent verification support for their process definitions. Furthermore, it proposes Petri nets as a means to fill this verification gap and argues why Petri nets are suitable for this.
Finally, it relates existing research on the verification of workflow process definitions to our work.

Chapter 3 introduces Petri nets. More specifically, it introduces the subclass of Petri nets we use in this thesis: the subclasses of P/T nets (simply called nets in this thesis) and WF-nets (workflow nets). It also defines on both classes the relevant Petri-net-related properties we need in the remainder of this thesis, and it introduces the property of soundness [Aal97, Aal98a, Aal00c] on WF-nets, and the life-cycle inheritance relation [Bas98, BA01, AB02] on sound WF-nets. As mentioned earlier, both the soundness property and the life-cycle inheritance relation are pivotal to Woflan and to this thesis.

Chapter 4 deals explicitly with the soundness property. Simplified, this property states that every WF-net can successfully reach the end. This chapter answers questions like

- How can this property be computed and
- What kind of diagnostic results can we obtain to repair any errors?

To do so, this chapter exploits the relation between the soundness property and two existing and well-known properties in the Petri-net field: liveness and boundedness. Finally, this chapter also includes a small case study to show that the diagnosis process it introduces is viable.

Chapter 5 deals with the computation of the life-cycle inheritance relation. As mentioned earlier, the complexity of a straightforward computation for life-cycle inheritance contains an additional exponential factor (when compared to the other three inheritance relations). This chapter shows that we can lessen this complexity problem to a large extent.

Chapter 6 deals with the question whether the subclass of WF-nets is expressive enough to model arbitrary workflow process definitions. To answer this question, we use the inter-organizational workflow process language XRL [AK02, VAK04], and show that we can map arbitrary XRL process definitions onto WF-nets. Because XRL supports the most relevant workflow patterns [AHKB00, AHKB03], it is more expressive than most existing workflow process languages, which typically only support a small subset of these patterns. Furthermore, this chapter includes interesting verification results for a (well-structured) language like XRL.

Chapter 7 strengthens the result of Chapter 6 by showing that workflow process definitions of market leaders like IBM MQSeries Workflow (or WebSphere MQ Workflow as it is called now), Staffware, and COSA Workflow can be mapped onto WF-nets.
Chapter 8 introduces the workflow verification tool Woflan and discusses its architecture and design decisions made. Woflan embeds the efficient and effective routines necessary to:

- decide soundness on WF-nets (Chapter 4),
- decide any of the four inheritance relations between sound WF-nets, among which life-cycle inheritance (Chapter 5), and
- map existing workflow process definitions onto WF-nets (Chapter 6 and Chapter 7).

Based on these routines, a Windows application and a Web service application have been built.

Chapter 9 embeds a number of case studies that show that the entire approach of using Petri nets as a means for workflow verification is viable. Two case studies show that the diagnosis process as defined in Chapter 4 can be used effectively to diagnose and correct erroneous workflow process definitions, including a complex Staffware process definition, while a third case study shows that the algorithm presented in Chapter 5 can substantially improve processing times when checking life-cycle inheritance.

Chapter 10 contains concluding remarks discussing the results of this thesis, and relates these results to known alternatives to Woflan.

For sake of completeness, this thesis contains four appendices. Appendix A contains an exhaustive lists of properties that can be computed with and retrieved from Woflan. Appendix B contains the full description of the TPN file format, which has been Woflan’s native file format since the beginning. Appendix C contains the full description of a travel agency, which is used as a case study in Chapter 4. Finally, Appendix D contains the Data Type Definition (DTD) and a reference to the XML Schema Definition (XSD) of XRL, which is used as a proof-of-concept for the use of WF-nets in Chapter 6.
CHAPTER 2

**Workflow management**

“It is in self-limitation that a master first shows himself.”

Johann Wolfgang von Goethe

As the previous chapter states, this thesis presents both theoretical and practical results that have been used to develop the tool Woflan, and that Woflan is a tool that can verify workflow process definitions. This chapter introduces workflow management and shows that the design of a workflow process definition can be far from trivial. As a result, it is reasonable to assume that errors are made. Unfortunately, until now, verification of workflow process definitions has received little to no attention. As a result, there is almost no tool support to detect whether an error is made.

First, we introduce the workflow concepts we need for the remainder of this thesis. Second, we argue that, for this thesis, it is reasonable to restrict ourselves to workflow process definitions. Third, we introduce two properties we think are pivotal when verifying workflow process definitions. Fourth, we introduce the approach we plan to take for verifying these properties. Fifth, we show that this thesis builds on earlier publications and comment on related work. Last, we conclude this chapter.

### 2.1 Workflow

In the last decade, workflow management systems have become a popular tool to support the logistics of business processes in banks, insurance companies, and governmental institutions [HL91, JB96, Kou95, Law97, Sch96, She97, Aal98a, Fis01, AH02]. Before, there were no generic tools to support workflow management. As a result, parts of the business process were hard-coded in the applications. For example,
an application to support task Notify approver (see Figures 1.1, 1.2, and 1.3 on pages 3 to 5) triggers another application to support task Verify authority. This means that one application knows about the existence of another application. This is undesirable, because every time the underlying business process is changed, applications need to be modified. Moreover, similar constructs need to be implemented in several applications and it is not possible to monitor and control the entire workflow. Therefore, several software vendors recognized the need for workflow management systems. A workflow management system is a generic software tool that allows for the definition, execution, registration, and control of business processes or workflows. At the moment, many vendors are offering a workflow management system. This shows that the software industry recognizes the potential of workflow management tools.

The fundamental property of a workflow process is that it is case-based. This means that every piece of work is executed for a specific case, also called a workflow instance. Examples of cases are a requisition request, an insurance claim, a tax declaration, a customer complaint, a mortgage, an order, or a request for information. Thus, handling a requisition request, an insurance claim, a tax declaration, or a customer complaint are typical examples of workflow processes. A typical example of a process that is not case-based, and hence not a workflow process, is a production process such as the assembly of bicycles. The task of putting a tire on a wheel is (generally) independent of the specific bicycle for which the wheel will be used. Note that the production of bicycles to order, that is, procurement, production, and assembly are driven by individual orders, can be considered as a workflow process.

The goal of workflow management is to handle cases as efficient and effective as possible. A workflow process is designed to handle large numbers of similar cases. Handling one requisition request usually does not differ much from handling another requisition request. The basis of a workflow process is the workflow process definition. This process definition specifies which tasks need to be executed in what order. Alternative terms for workflow process definition are: ‘procedure’, ‘workflow schema’, ‘flow diagram’, and ‘routing definition’. Tasks are ordered by specifying for each task the conditions that need to be fulfilled before it may be executed. In addition, it is specified which conditions are fulfilled by executing a specific task. Thus, a partial ordering of tasks is obtained. Figures 1.1, 1.2, and 1.3 give such partial orderings in a graphical way. In a workflow process definition, standard routing elements are used to describe sequential, alternative, parallel, and iterative routing thus specifying the appropriate route of a case. The workflow management coalition (WFMC) has standardized a few basic building blocks for constructing workflow process defini-
A so-called **OR-split** is used to specify a choice between several alternatives; an **OR-join** specifies that several alternatives in the workflow process definition come together. Please note that OR-splits and OR-joins may be non-exclusive, in which case multiple alternatives (possibly all) can be chosen or need to be synchronized. To distinguish non-exclusive choices from exclusive choices, we also introduce **XOR-splits** and **XOR-joins**. An **XOR-split** is used to specify an exclusive choice between several alternatives; an **XOR-join** specifies that several alternatives in the workflow process definition come together without any synchronization. Examples of XOR-splits are the tasks Select approver, Notify approver, and Verify authority in Figures 1.1 and 1.2 and the routing constructs Approver found?, Request approved?, and Spending limit OK? in Figure 1.3. Examples of XOR-joins are the routing construct Rejected in Figure 1.2 and the tasks Select approver, Notify approver, and Record result in Figure 1.3. An **AND-split** and an **AND-join** can be used to specify the beginning and the end of parallel branches in the workflow process definition. Examples of AND-splits are the routing construct AND split in Figures 1.1 and 1.2 and the routing construct Approver found? in Figure 1.3 in case an approver is found. Examples of AND-joins are the routing construct AND join in Figures 1.1 and 1.2 and the hour glass object to the left of task Notify approver in Figure 1.3. The routing decisions in OR-splits are often based on data such as the age of a customer, the department responsible, or the contents of a letter from the customer. In case of the OR-split Approver found? the decision is based on the fact whether the requestor has a manager.

Many cases can be handled by following the same workflow process definition. As a result, the same task has to be executed for many cases. A task that needs to be executed for a specific case is called a **work item**. An example of a work item is the order to execute task Select approver for case ‘Requisition request of employee baker for a flat-panel’. Most work items need a **resource** in order to be executed. A resource is either a machine (that is, a printer or a fax) or a person (participant, worker, or employee). Besides a resource, a work item often needs a **trigger**. A trigger specifies who or what initiates the execution of a work item. Often, the trigger for a work item is the resource that must execute the work item. Other common triggers are external triggers and time triggers. An example of an external trigger is an incoming phone call of a customer; an example of a time trigger is the expiration of a deadline. See, for example Figures 1.1, 1.2, and 1.3 where a deadline is associated with the task Notify approver. A work item that is being executed is called an **activity**. If we take a photograph of the state of a workflow, we see cases, work items, and activities. Work items link cases and tasks. Activities link cases, tasks, triggers, and resources. Figure 2.1 shows this in a graphical way, whereas
Figure 2.2 shows how these concepts can be positioned in a workflow meta-model.

The meta-model of Figure 2.2 shows that at build-time the designer models a process like the requisition process in Figure 1.3, that contains tasks, like Select approver. A task is related to other tasks through transitions, and tasks may either be atomic or correspond to an entire subprocess. In case of complex task relations, routing constructs like Approver found? or the hour glass may be needed, but this is not reflected explicitly in the meta model, as these routing constructs are considered to be part of the corresponding transitions. A transition may have a condition, which may depend on data. For example, the transition from task Notify approver to task Verify authority has the condition approved="Yes", where approved is a data element. A task can be performed by some role corresponding to some team, and any number of applications can be associated with the task. An application may need and/or modify data. For example, the task Notify approver needs to set the data element approved to either “Yes” or “No”. At run-time, multiple cases may correspond to a process definition. Depending on the current state of the case, a number of work items is associated with the case. A work item corresponds to a task that can be performed by some resource, and multiple triggers can be associated to it. As soon as a resource starts the work item, the work item becomes an activity. To an activity, the actual resource that performs it is associated, together with a number of application instances, that may need and/or modify data values as specified at build-time.
2.2 Abstractions

Workflow management has many aspects and typically involves many disciplines. The issues presented in this thesis focus on the control-flow perspective (that is, on the workflow process definitions) and abstract from other perspectives. This section motivates why it is reasonable to restrict the analysis focus to a single perspective. Therefore, we start by introducing the perspectives commonly identified in workflow literature.

2.2.1 Perspectives

The primary task of a workflow management system is to enact case-driven business processes by joining several perspectives. The follow-
Workflow management

...ing perspectives are relevant for workflow modeling and workflow execution:

1. control-flow (or process) perspective,

2. resource (or organization) perspective,

3. data (or information) perspective,

4. task (or function) perspective,

5. operation (or application) perspective.

(These perspectives are similar to the ones given in [JB96].)

In the control-flow perspective, workflow process definitions are defined to specify which tasks need to be executed and in what order (that is, the routing or control flow). The concepts relevant for this perspective (task, condition, and AND/OR-split/join) have already been introduced in Section 2.1.

In the resource perspective, the organizational structure and the population are specified. Resources, ranging from humans to devices, form the organizational population and are mapped onto resource classes. In office environments, where workflow management systems are typically used, the resources are mainly human. However, because workflow management is not restricted to offices, we prefer the term resource. To facilitate the allocation of work items to resources, resources are grouped into classes. A resource class is a group of resources with similar characteristics. There may be many resources in the same class and a resource may be a member of multiple resource classes. If a resource class is based on the capabilities (that is, on the functional requirements) of its members, it is called a role. If the classification is based on the structure of the organization, such a resource class is called an organizational unit (that is, a team, a branch, or a department). The resource classification describes the structure of the organization.

The data perspective deals with control and production data. Control data are data introduced solely for workflow management purposes. Control data are often used for routing decisions in OR-splits. Production data are information objects (for example, documents, forms, and tables) whose existence does not depend on workflow management.

The task perspective describes the content of the process steps, that is, it describes the characteristics of each task. A task is a logical unit of work with characteristics such as the set of operations that need to be performed, description, expected duration, due-date, priority, trigger (that is, time, resource, or external trigger), and required resources...
Abstractions

(classes (that is, roles and organizational units). For sake of readability, in Figure 2.2 we only mention triggers and resource classes.

In the operational perspective, the elementary actions are described. Note that one task may involve several operations. These operations are often executed using applications ranging from a text editor to custom-built applications for performing complex calculations. Typically, these applications create, read, or modify control and production data in the data perspective.

This thesis addresses the problem of qualitative workflow verification. That is, we focus on properties of a logical nature and not on performance issues (quantitative analysis). For the purpose of qualitative verification, we only consider the control-flow perspective of a workflow. Figure 2.3 shows this perspective. In the remainder of this section, we
discuss a number of abstractions motivating why this simplification is reasonable.

2.2.2 Abstraction from resources

Detailed knowledge of the allocation of resources to work items, the duration of activities, and the timing characteristics of triggers are a crucial factor when analyzing the performance of a workflow. However, for qualitative verification, it is only relevant whether certain execution paths are possible or not. It is important to note that the allocation of resources can only restrict the routing of cases, that is, it does not enable execution paths that are excluded in the control-flow perspective. Since resource allocation can only exclude execution paths, for qualitative verification, it suffices to focus on potential deadlocks resulting from the unavailability of resources. In the next few paragraphs, we argue that deadlocks resulting from restrictions imposed by resource allocation are generally absent, thus motivating why it is reasonable to abstract from resources.

A potential, resource-inflicted deadlock could arise

1. when multiple tasks try to allocate multiple resources at the same time, or
2. when there are tasks imposing such demanding constraints that no resource qualifies.

The first type of deadlock often occurs in flexible manufacturing systems where both space and tools are needed to complete operations thus potentially resulting in locking problems [SV90]. However, given today’s workflow technology, such deadlocks cannot occur in a workflow management system: At any time, there is only one resource working on a task which is being executed for a specific case. As far as we know, in today’s workflow management systems, it is not possible to specify that several resources are collaborating in executing a task. Note that even if multiple persons are contributing to the execution of one activity, for example, writing a report for a given case, only one person is assigned to that activity from the perspective of the workflow management system: This is the person that selected the corresponding work item from the in-basket (that is, from the electronic worktray). Therefore, from the viewpoint of qualitative verification, it is reasonable to abstract from these locking problems. (Nevertheless, if in the future collaborative features are explicitly supported by workflow management systems, then these problems should be taken into account.)

The second type of deadlock occurs when there is no suitable resource to execute a task for a given case, for example, if there is not a single
Abstractions

resource within a resource class. Generally, such problems can be avoided quite easily by checking whether all resource allocations yield non-empty sets of qualified resources. However, there may be some subtle errors resulting from case management (a subset of tasks for a given case is required to be executed by the same resource) and function separation (two tasks are not to be executed by the same resource to avoid security violations). For example, task 1 should be executed by the same person as task 2 and task 2 should be executed by the same person as task 3. However, task 3 should not be executed by the person who executed task 1. Clearly, there is no person qualified to execute task 3. Such problems highly depend on the workflow management system being used and are fairly independent of the routing structure. Therefore, in our approach of workflow-product-independent verification we abstract from this type of resource-driven deadlocks.

2.2.3 Abstraction from data and triggers

Recall that the data perspective deals with both control and production data. We abstract from production data because these are outside the scope of the workflow management system. These data can be changed at any time without notifying the workflow management system. In fact, their existence does not even depend upon the workflow application and they may be shared among different workflow processes, for example, the bill-of-material in manufacturing is shared by production, procurement, sales, and quality-control processes.

We partly abstract from control data. In contrast to production data, the control data used by the workflow management system for routing cases are managed by the workflow management system. However, some of these data are set or updated by humans or applications. For example, a decision is made by a manager based on intuition or a case is classified based on a complex calculation involving production data. Clearly, the behavior of a human or a complex application cannot be modeled completely. Therefore, some abstraction is needed when verifying a given workflow. The abstraction used in this thesis is the following. Since control data are only used for the routing of a case, we incorporate the routing decisions but not the actual data. For example, the decision to accept or to reject a requisition request is taken into account, but not the actual data where this decision is based on. Therefore, we consider each choice to be a non-deterministic one. Moreover, we assume a fair behavior with respect to these choices and exclude conspiracies [Bes84]. Fair behavior excludes the possibility that in the requisition process the task Remind approver is executed over and over again: At some point, the approver will make a decision and the requisition request will be forwarded.
Workflow management

We also abstract from triggers, because a workflow management system cannot control the occurrence of triggers. As for choices, we only assume fairness with respect to the occurrence of triggers: An enabled task cannot be blocked forever (or infinitely often) because the corresponding trigger is never received.

The fairness assumptions on choices and triggers are reasonable: Without these assumptions any iteration or trigger would create a potential livelock or deadlock.

2.2.4 Abstraction from task content and operations

As a final abstraction, we consider tasks to be atomic, abstracting from the duration of tasks and the execution of operations inside tasks. The workflow management system can only launch applications or trigger people and monitor the results. It cannot control the actual execution of the task. Therefore, from the viewpoint of qualitative verification, it is reasonable to consider tasks as atomic entities.

Note that we do not explicitly consider transactional workflows [GHS95]. There are several reasons for this. First of all, most workflow management systems (in particular the commercial ones) do not support transactional features in the workflow modeling language. Second, as is shown in [AAH98], the various transactional dependencies can easily be modeled in terms of Petri nets. Therefore, we can straightforwardly extend the approach in this thesis to transactional workflows.

2.3 Verification issues

Workflow process definitions can become very complex, especially when a workflow process spans several organizations (inter-organizational workflow processes). Because of the complexity, the designers of such a workflow process definition can easily introduce errors. Unfortunately, today’s workflow management systems do not detect many errors and, hence, cannot guide the designer towards correcting these errors.

When for a complex inter-organizational workflow [LB96, DK01, Aal00a, Aal00b, GAHL01, KZ02] different designers from different organizations are responsible for different parts of the workflow process, it gets even more complicated. Typically, in such a situation, every designer has to design a workflow process that corresponds to the part of the workflow process that belongs to his organization. To ensure that his workflow process will fit with the other workflow pro-
cess definitions, his definition has to adhere to some specification. If all do, the entire workflow process will work, if one of them does not, the entire workflow process may fail, leading to a small catastrophe for all organizations involved. Therefore, the designers of such workflow process definitions need to be able to check whether their definitions adhere to the specifications that were agreed.

To further complicate matters, workflow management systems need to be flexible [WR96, KDB98, She97, CCPP98, ABV+00]. However, running instances might be involved when trying to modify a workflow process definition. Possibly, these running instances need to migrate from the old workflow process to the new workflow process. Thus, the new workflow process definition should be such that it embeds all possible behavior of the old workflow process definition, and a designer should be able to check this.

To be able to handle these complex situations, we propose to build a verification tool that can verify the following properties:

- a workflow process definition in isolation is sound [Aal97, Aal98a, Aal98c], and
- any of the four inheritance relations exists between one workflow process definition and a second workflow process definition [AB97, Bas98, BA01, AB02].

Using the soundness property, we can check whether the entire workflow process definition satisfies some minimal requirements (option to complete, proper completion, and no dead tasks), while using the inheritance relations we can check whether:

- a workflow process definition adheres to some specification that was agreed beforehand, and
- a new workflow process definition embeds the behavior of the old workflow process definition, thus all running instances can safely migrate from the old to the new definition.

In the following subsections, we explain these properties in some detail.

### 2.3.1 Soundness

In [Ade01], Ader compares a number of existing workflow management systems, using a number of criteria, which he subdivides into features. Among these features is the Process Verification feature, which he defines as follows [Ade01] (the numbers between brackets signify the scores per feature):
A set of features to verify the correct definition of a process, thus avoiding the discovery of errors at test time, or worse still, at run time.

**Missing elements** (5), prevents the definition tool from delivering a process when elements are needed for its normal execution are missing. This is the minimal feature.

**Type checking** (3), check type compatibility for all variables usages and value assignment. This is a very important feature that implies a form of compilation of defined script even if this is not required by the script interpreter.

**Dead ends** (2), detection by analysis of the process description of parts of the network that will never be used.

**Deadlocks** (2), detection through network analysis of situations where parallel parts may enter in a deadlock situation.

**Step-by-step execution** (3), is essential for debugging process definitions.

**Score 0 to 15**

From the previous section, it will be clear that our focus is on the third and fourth feature: dead ends and deadlocks. However, we believe that these features insufficiently guarantee the absence of control-flow errors. In our eyes, it is crucial to verify that a workflow process definition in isolation exhibits only proper behavior. Pivotal to this is the notion of **soundness** [Aal97, Aal98a, Aal98c], which expresses minimal requirements any workflow process definition should satisfy.

Recall that a workflow process definition is sound if it satisfies the following conditions: option to complete, proper completion, and no dead tasks. The option-to-complete condition extends the Deadlocks feature as mentioned by Ader, whereas the no dead tasks condition is equivalent to Ader’s Dead ends feature. Thus, the features as mentioned by Ader do not detect livelock (work is being done on a case, but the case cannot be completed), nor can they detect improper completion (work is being done on a case, while the case has been completed).

### 2.3.2 Inheritance of behavior

In the literature, several formalizations of what it means for a workflow process definition to extend the behavior of another definition have been studied; see [BA01] for an overview. An interesting option in this context is to use the concept of inheritance. The inheritance mechanism allows the designer to specify a class, the subclass, that inherits features of some other class, its superclass. Thus, it is possible
to specify that the subclass has the same features as the superclass, but that in addition it may have some other features.

The informal definition of inheritance in UML [RJB98, BRJ98, Gro] states the following:

The mechanism by which more specific elements incorporate structure and behavior defined by more general elements. [BRJ98].

However, only the class diagrams, describing purely structural aspects of a class, are equipped in UML with a concrete notion of inheritance. It is implicitly assumed that the behavior of the objects of a subclass as defined by the workflow process definition, is an extension of the behavior of the objects of its superclass.

A possible formalization of behavioral inheritance is called life-cycle inheritance [AB97, Bas98, BA01, AB02]:

A workflow process definition is a subclass of another workflow process definition under life-cycle inheritance if and only if it is not possible to distinguish the external behavior of both when the new methods, that is, the methods only present in the potential subclass, are either blocked or hidden.

The notion of life-cycle inheritance has been shown to be a sound and widely applicable concept. In [BA01], it has been shown that it captures extensions of workflow process definitions through common constructs such as parallelism, choices, sequencing and iteration. In [AB01], it is shown how life-cycle inheritance can be used to analyze the differences and the commonalities in sets of workflow process definitions, for example, between a process and its potential successor (after some necessary modifications were carried out.) Furthermore, in [Aal02a], the notion of life-cycle inheritance has been successfully lifted to the various behavioral diagram techniques of UML. Finally, life-cycle inheritance can be used to tackle problems related to dynamic change of workflow processes [AB02]; furthermore, it has proven to be useful in producing correct interorganizational workflows [Aal02b].

2.4 Approach

In a previous section, we showed that for the purpose of qualitative verification it is reasonable to abstract from resources, data, triggers, the content of tasks, and operations and to focus on the control-flow perspective. In fact, it suffices to consider the control flow of one case
Workflow management

in isolation. The only way cases interact directly, is via the competition for resources and the sharing of production data. (Note that control data are strictly separated.) Therefore, if we abstract from resources and production data, it suffices to consider one case in isolation. The competition between cases for resources is only relevant for performance analysis.

The principal goal of the approach presented in this thesis is to verify the correctness of a workflow specified in some workflow management system, that is, the approach is not tailored towards a specific workflow management system.

Despite the efforts of the Workflow Management Coalition (WfMC, [WFM96, Law97, Fis01]), there is still no consensus on the language for specifying workflows. For exchanging workflow process definitions, the WfMC proposes either the Workflow Process Definition Language (WPDL) [WFMa] or the XML Process Definition Language (XPDL) [WFMb]. However, both are only partially supported by the existing systems. (Typically, workflow management systems are unable to import and handle all constructs.) Moreover, both have no formal semantics [Kie02, Aal03a] which means that it is impossible to reason about the correctness of a given workflow process definition.

Based on the above mentioned lack of consensus, we decided to base our approach on Petri nets [Pet81, Rei85, Mur89, RR98a, RR98b]. Petri nets are a universal modeling language with a solid foundation. Yet, Petri nets are close to the diagramming techniques used in today’s workflow management systems [MEM94, ADE98, Aal98a, ADO00, AH02, AHW03]. The efficient analysis techniques developed for Petri nets over the last 40 years [Bes84, CS90, ES90, Fin93, Val94, BCD95, Var98, KA99] allow for the analysis of complex workflows. The graphical representation of Petri nets and the available analysis techniques are particularly useful for generating diagnostic information. As indicated in [Aal98a, AAH98, EN93], they are a good starting point for a solid foundation of workflow management. Therefore, we propose to directly map workflow process definitions specified in existing workflow management systems onto WF-nets (workflow nets, a subclass of Petri nets), applying the abstraction discussed in Section 2.2. The resulting WF-nets should of course be consistent with the (formal or informal) semantics of the workflow process definitions as defined by the workflow management systems being used.

Although we base our approach on WF-nets, we do not assume that workflow management systems are based on WF-nets too. Instead, we propose the approach illustrated by Figure 2.4. As Figure 2.4 shows, there is a specific mapper for each workflow management system. Such a mapper maps a workflow process definition onto a WF-net.
During the mapping, the abstraction discussed in Section 2.2 is used to extract the information required for qualitative verification. It is important to note that the workflow verification tool is \textit{not} used to edit the workflow process definition. If the verification tool detects errors, then the diagnostics provided by the verification tool are used to correct the errors using the design tools of the workflow management system itself. As Figure 2.4 shows, the process of correcting the errors is iterative: The workflow process definition constructed using the workflow management system is mapped and analyzed using the verification tool. Then, the diagnostics are used to correct (if necessary) the process definition using the workflow management system. This procedure is repeated until all errors have been repaired.

The approach illustrated in Figure 2.4 stands or falls with the assumption that the diagnostics are of high-quality and workflow-system independent. Since most workflow management systems model workflows in terms of a graph structure connecting tasks, it is possible to make the diagnostics relatively system independent. For example, the verification tool can present a list of tasks which cannot be executed or show...
Workflow management

Execution sequences in terms of tasks which lead to a deadlock. These diagnostics can be interpreted in the context of any workflow management system. To improve the feedback to the workflow modeler, it is possible to use the diagnostics to highlight the errors directly in the design tools of workflow management systems. Note that the latter requires extensions of the workflow management system itself.

2.5 Related work

The previous sections introduced workflow management, argued that for qualitative verification we can restrict ourselves to the control-flow perspective of workflow (that is, on the workflow process definitions), presented relevant verification issues, proposed to use Petri nets as our formalism for verification, and showed the approach we propose. These sections have already discussed some related work into some detail. This section will relate other work to this thesis.

2.5.1 Workflow management systems

A lot has been written on workflow management systems in general, which makes it impossible to give a complete overview of existing literature. For an overview on some workflow management systems, we refer to Ader [Ade01], which presents a comparative study between the workflow management systems shown in Table 2.1. Although Ader does not focus on the control-flow perspective, some conclusions can be drawn from this study. First, verification of workflow process definitions is included in the study, but this verification only includes dead tasks and deadlocks (which are both included in the soundness property). Second, although dead ends and deadlocks are included in the study, their importance is considered only minor: they score less than 3% of the total score of the second-most important feature (process power), where the most important feature is throughput rates.

<table>
<thead>
<tr>
<th>BizFlow (Handysoft)</th>
<th>Staffware (Staffware corp.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>COSA Workflow (COSA Solutions)</td>
<td>TeamWARE Flow (Fujitsu)</td>
</tr>
<tr>
<td>DOLPHIN (Fujitsu)</td>
<td>TIB/InConcert (TIBCO)</td>
</tr>
<tr>
<td>Eastman Software Enterprise Workflow</td>
<td>Visual and Panagon Workflow (FileNet)</td>
</tr>
<tr>
<td>InTempo (JetForm)</td>
<td>W4 (W4)</td>
</tr>
<tr>
<td>MQSeries Workflow (IBM)</td>
<td>WFX (Eastman Software)</td>
</tr>
<tr>
<td>SERFloware (SER)</td>
<td></td>
</tr>
</tbody>
</table>

**TABLE 2.1. Workflow management systems compared in [Ade01].**
For the remainder, and because the focus of this thesis on process definition tools, we restrict ourselves to workflow management systems literature with a strong focus on the process definition tools, as these are relevant when mapping their process definitions onto WF-nets. For information on the Staffware Process Definer in specific, we refer to [Sta97, Sta99, Sta02]. Although a lot of workflow management systems have been (and still are) on the market, Staffware has been a major player. Another major player is IBM with its MQSeries workflow (nowadays called IBM WebSphere MQ Workflow) [Köh03, WAB01, IBM01a, IBM01b, IBM99, IBM01c]. In the Netherlands, the Petri-net-based system COSA Workflow [SL98, Baa99] is also a major player. Other players include the ad hoc system TIBCO InConcert [TIB00], Eastman [Sof98], the case handling system FLOWer [Ath02], Lotus Domino [NEG+00], Forté [Sun00], and Verve [Ver00]. Research prototypes like ADEPTflex [RD98], EPCs [KNS92], Mentor [WWW99], Meteor [SKM], Mobile [JB96], WASA [Wes00], XRL/Flower [VHA02], and Yawl [AH03] also exist. Last, we mention that, for example, SAP [SAP00] also has a workflow module.

In the areas of business process management and workflow management in general, standard publications include [GHS95, Kou95, JB96, Sch96, Law97, ADE98, ADO00, Fis01, AH02, AHW03].

### 2.5.2 Workflow process definitions

These workflow management systems all use some proprietary formalism to manage their process definitions. A lot of research has been done on suitable ‘best-practice’ formalisms for workflow process definitions. We already have discussed WfMC’s WPDL [WMa] and XPDL [WMb], and decided to use Petri nets, like more researchers have done [Aal96, Aal98a, Aal98b, AAH98, ED03]. However, other researchers have proposed alternatives, for example,

- Eshuis [EW02, Esh02] proposes to use the Unified Modelling Language (UML),
- Dehnert [Deh03] proposes to use Event-driven Process Chains (EPC’s) [Sch94, Aal99], and
- Schroeder [Sch99] proposes the Process Interchange Format (PIF) [LGJ+98].

Another line of research is on patterns that typically occur in workflow process definitions [AHKB00, Aal03b, AHKB03, KHA03]. Using this line of research it is possible to compare different languages for workflow process definitions. In contrast with [Ade01], [AHKB03] contains such a comparative study which explicitly focuses on the control-flow perspective of the workflow management systems listed in Table 2.2. Note that the inter-organizational workflow language XRL, which is
used in Chapter 6 to show that our WF-net-based approach is applicable in general, is based on these patterns. Work that is closely related to the patterns research is the research on expressiveness of workflow languages [Kie02].

Given the fact that flexibility is an important open issue in workflow management, research in this area also exists. Examples are the research done on ADEPTflex [RD98, RRD03b, RRD03a] and on process inheritance [Aal02a, Aal02b, AB02, AB01].

2.5.3 Verification of workflow process definitions

Only a few papers in the literature focus on the verification of workflow process definitions. In [HOR98], some verification issues have been examined and the complexity of selected correctness issues has been identified, but no concrete verification procedures have been suggested. In [Aal97, Aal00c, AAH98], concrete verification procedures based on Petri nets have been proposed. As mentioned earlier, our work builds upon the techniques presented in [Aal97, Aal00c]. The technique presented in [AAH98] has been developed for checking the consistency of transactional workflows including temporal constraints. However, the technique is restricted to acyclic workflows and only gives necessary conditions (that is, not sufficient conditions) for consistency.

As far as we know, only a few other tools have been developed for verifying workflows. One of them is FlowMake [SO, SO99, SO00]. To our knowledge, FlowMake has been the only other verification tool for some years. It is a tool based on the reduction technique described in [SO99] and can interface with the IBM MQSeries Workflow product. FlowMake can only handle acyclic workflows and provides only the reduced workflow graph as diagnostic information. The reduction technique uses a correctness criterion which corresponds to soundness.

### Table 2.2. Workflow management systems compared in [AHKB03]

<table>
<thead>
<tr>
<th>System</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Changengine (HP)</td>
<td>Mobile (Univ. of Erlangen/Nürnberg, Germany)</td>
</tr>
<tr>
<td>COSA Workflow (COSA Solutions)</td>
<td>MQSeries Workflow (IBM)</td>
</tr>
<tr>
<td>Domino Workflow (Lotus/IBM)</td>
<td>SAP R/3 Workflow (SAP)</td>
</tr>
<tr>
<td>Eastman Software Enterprise Workflow</td>
<td>Staffware (Staffware plc)</td>
</tr>
<tr>
<td>FLOWer (Pallas Athena)</td>
<td>TIB/InConcert (TIBCO)</td>
</tr>
<tr>
<td>Forté Conductor (Forté)</td>
<td>Verve</td>
</tr>
<tr>
<td>I-Flow (Fujitsu)</td>
<td>Visual Workflo (FileNet)</td>
</tr>
<tr>
<td>Meteor (LSDIS, University of Georgia)</td>
<td></td>
</tr>
</tbody>
</table>
Related work

and the class of workflow processes considered are in essence acyclic free-choice Petri nets (see [DE95] for free-choice Petri nets). However, the set of reduction rules as suggested in [SO99] is not complete, as is shown in [LZLC02, AHV02]. [LZLC02] also proposes a set of rules which they claim to be complete, but for which the complexity is an order of magnitude worse than for the original set. Based on this, [AHV02] shows that using the well-known reduction rules for free-choice Petri nets from [DE95] has two advantages:

- The complexity is slightly better;
- Cyclic workflow processes are allowed.

Some work on the compositional verification of workflows, using other well-known Petri-net results such as the refinement rules in [Val79], can be found in [Aal00c, AB02, Voo00].

In recent years, some tools based on model checking [CGP99] appeared. Although Eshuis [Esh02, EW02] promotes the use of UML Activity Diagrams for modelling workflow process definitions, when it comes down to verifying these diagrams, he maps them onto finite transition systems and uses a model checker [CGP99] on these systems to verify whether they satisfy certain temporal logic formulas. If such a system does not satisfy some formula, the tool will provide a counter example for that formula. Using this counter example, the designer can repair the error. However, not every UML Activity Diagram corresponds to a finite transition system: some result in an infinite transition system. To deal with such diagrams, Eshuis proposes a finiteness check during the mapping which is well-known in the Petri-net community: the boundedness check in the Karp-Miller graph algorithm. Unfortunately, Eshuis does not mention how his tool can help the designer correcting the error in such cases. However, Eshuis’ approach is able to deal with data to a certain extent, whereas we totally abstract from data. However, to prevent the resulting transition system from becoming infinite, Eshuis restricts data expressions to expressions only containing the ∧, ∨, ¬, and = operators.

Other approaches also using model checking techniques are mentioned in, for instance, [KGMW, Sch99, Mad01, BFG+01, GR01, Mat03]. Besides the fact that Karamanolis et al. [KGMW] restrict their workflow process definitions to acyclic ones, these approaches also presuppose that the workflow process definitions correspond to finite transition systems. This is a general weakness in model checking approaches, because, as mentioned by Eshuis, workflow process definitions do not necessarily correspond to finite transitions systems.

Finally, an overview of verification problems in workflow process definition is given by Hofstede, Orłowska, and Rajapakse [HOR98].
2.5.4 Petri nets

The research on Petri nets originates from the thesis of Petri [Pet62], although the term “Petri nets” has been coined only afterwards. According to Murata [Mur89], until 1980 the research on Petri nets has been conducted mainly in the United States, for example at MIT. The first book on Petri nets was written by Peterson [Pet81], which contains a lot of annotations of all Petri-net-related research conducted before 1980. As from the late-1970’s, many Europeans and others started conducting research on Petri nets, which resulted in an impressive list of publications. At the moment, standard publications in the area of Petri nets include [Rei85, Mur89, EN94, DE95, RR98a, RR98b]. Publications on the application of Petri nets in the workflow management area include [Ell79, Aal96, AAH98, Aal98a, Aal98b, ADE98].

2.5.5 WF-nets

Like this thesis, the thesis of Dehnert [Deh03] is also based on the WF-net-related work of Van der Aalst [Aal97, Aal98a, Aal00c]. In her thesis, Dehnert argues that, while soundness is an appropriate property for workflow process definitions in the final stages of their design (on the implementation level), it is too strict in the early stages (on the specification level). Her argumentation is that in the early stages designers mainly focus on incorporating all possible proper behavior, while in the final stages their focus shift to preventing any improper behavior. For supporting the early stages, she introduces the concept of relaxed soundness and proposes to use EPC’s. For supporting the final stages, she proposes to use soundness and WF-nets. Basically, a workflow process definition is relaxed sound if all tasks are covered by proper-behaving scenarios. Although we do not use it elsewhere in this thesis, Woflan can verify relaxed soundness provided the WF-net corresponds to a finite transition system (for WF-nets that correspond to infinite transition systems, Woflan also will give an answer, but this answer might be negative while it should have been positive.)

Eshuis and Dehnert [ED03] use WF-nets except for the fact that they partition the set of transitions into a set of internal transitions and a set of external transitions, where firing an internal transition has precedence over firing an external transition. The resulting WF-nets are called reactive WF-nets.

Van Hee, Sidorova, and Voorhoeve [HSV03] use WF-nets to study several correctness notions and refinement rules preserving these correctness notions. Their most important correctness notion is called soundness, but does not entirely correspond to the notion of soundness used in this thesis. Where we define soundness using only a single case
in the system, Van Hee et al. define soundness on any number of cases in the system, and they call our notion of soundness 1-soundness.

Based on this generalized notion of soundness, Van der Toorn [Too04] uses WF-nets to capture the behavior of software components and their specifications, and projection and life-cycle inheritance to determine whether the behavior of such a component matches the behavior of its specification.

2.5.6 Base of this thesis

The majority of this thesis is based on existing own work: Part of this chapter, Chapter 3, Chapter 4, Chapter 8, and part of Chapter 9 are based on [VBA01, VT04], Chapter 6 is based on [VAK04], while the remainder of Chapter 3 and Chapter 5 are based on [VB03].

[VBA01] concerns the verification of workflow process definitions using the tool Woflan, and is based on work of Van der Aalst on the interplay between verification of workflow process definitions and Petri nets, that is, on [Aal97, Aal98a, Aal00c]. [VBA01] extends these publications with:

- relations between the three requirements on soundness on the one hand and boundedness and liveness on the other,
- a diagnosis process,
- behavioral error sequences,
- a Woflan tool, and
- experimental results.

[VAK04] concerns the verification of XRL, also gives the formal Petri-net-based semantics of this (academic) inter-organizational workflow language, and is based on work of Van der Aalst, Kumar, and Verbeek [AK02, AVK01a, AVK01b]. In [AK02], Van der Aalst and Kumar introduced the inter-organizational workflow language called XRL and provided it with an informal Petri-net-based semantics. [AVK01a] presents some verification results using this first semantics, but needs to introduce a concept of extended WF-nets to be able to get verification results and extend the Woflan tool accordingly. [AVK01b] recognizes this and sketches a WF-net-based semantics for XRL. [VAK04] accumulates these publications by giving the complete and formal WF-net-based semantics and relevant verification results. Another XRL-related publication is [VHA02], which introduces the idea to build a Petri-net-based workflow system, called XRL/Flower, which uses XRL as its native workflow process definition language. This system uses the previously mentioned WF-net-based semantics to map any XRL instance onto a WF-net, and successively starts enacting that WF-net. Advantages of doing so are that we can use Woflan to verify any XRL
instance before enacting it, and that we can extend XRL as long as we can map XRL onto WF-nets (thus, we do not have to extend the Petri-net-based enactment server.)

[VB03] concerns deciding the life-cycle inheritance relation between two sound workflow process definitions and is based on the already mentioned work of Basten and Van der Aalst [AB97, Bas98, BA01, AB02]. In [AB97, Bas98], Van der Aalst and Basten introduce four inheritance relations, among which the life-cycle inheritance relation. [BA01] can be considered as a revised version of [AB97], whereas [AB02] applies these four inheritance relations to the workflow field. Of these four inheritance relations, life-cycle inheritance is the most complex to compute (due to an additional exponential factor). [VB03] alleviates this additional complexity problem to a large extent, whereas [VT04] concerns a case study which shows that the algorithm presented in [VB03] indeed alleviates excessive processing times.

Parts of Chapter 7 are based on existing work of other researchers: The Staffware mapping is based on [AH00], whereas the MQSeries Workflow mapping is based on [LR99].

[AH00] concerns the mapping from workflow task structures onto Petri nets and is based on [Aal97, Aal98a, HOR98]. Our mapping of Staffware is based on this, but extends it with subprocesses and Staffware’s withdraw functionality.

In [LR99], the semantics of IBM’s MQSeries Workflow processes is described in detail. From this, we extracted the semantics for its workflow process definitions, and based our mapping on this. In Chapter 5 of [LR99], Leymann and Roller describe possible extensions to the current MQSeries Workflow semantics. We did not take these extensions into account into our current mapping, as they are not operational yet.

2.6 Conclusions

Today, a lot of workflow management systems exist. Unfortunately, every workflow management system uses a different way to model workflow process definitions, and the issue of verifying workflow process definitions has been grossly neglected. Today’s workflow processes become more and more complex, especially because they often span more than one organization (inter-organizational workflow processes). If the part of the workflow process in one organization fails, then it might have a serious effects on the other organizations involved.
Still, verification has been neglected from the start and even today throughput issues dominate verification issues [Ade01].

Our approach is to map workflow process definitions onto WF-nets, to check whether a WF-net in isolation is sound (completion is guaranteed, completion is always proper, and every task has a meaningful role in the workflow process), and to check whether a WF-net extends the behavior of another WF-net (a modified workflow process definition should embed the behavior of its original). Using these properties, workflow process definitions can be checked automatically, and the designer of an erroneous definition can be guided towards correcting errors.

We restricted our approach to the control-flow perspective of workflow process definitions. Most importantly, we abstract partly from the data perspective, even though data might be used to control decisions in a workflow process definition. As result of this abstraction, decisions that depend on the same data element, and are thus dependent, now become independent, which leads to additional behavior. Nevertheless, we think that the decision to abstract from data is justified. First of all, one can argue that data (including control data) is volatile. Therefore, one should not use rely on data to exclude undesired behavior. Second, by using data to exclude undesired behavior one relates different decisions in an implicit way, where it would have been better to make this relation explicit in the workflow process definition.

As a result, if data is used to rule out unwanted behavior, soundness of a workflow process definition might not exactly correspond to soundness of the corresponding WF-net. Therefore, we investigate what possible discrepancies exist, based on the three soundness requirements (option to complete, proper completion, and no dead tasks), and using the observation that by abstracting from data, the behavior of the corresponding WF-net possibly extends (but does not restrict) the behavior of the workflow process definition.

**Option to complete.** If the corresponding WF-net has the option to complete, then the workflow process definition has the option to complete too: Every reachable state in the workflow process definition corresponds to some reachable state in the WF-net, and from that state completion is possible.

**Proper completion.** If completion of the corresponding WF-net is always proper, then completion of the workflow process definition is always proper: Every reachable state in the workflow process definition corresponds to some reachable state in the WF-net, therefore, improper completion of the workflow process definition is impossible.
No dead tasks. If the workflow process definition contains no dead tasks, then the corresponding WF-net contains no dead tasks (transitions): Every reachable state in the workflow process definition corresponds to some reachable state in the WF-net, thus, for a task enabled in the workflow process definition corresponds to a transition enabled in the WF-net.

Thus, because we abstract from data, if a WF-net satisfies the first two requirements of soundness (option to complete and proper completion), then so will the workflow process definition. However, if a WF-net satisfies the third requirement (no dead tasks), the workflow process definition might still contain tasks that are dead because of data.

Besides our approach, only a workflow-graph-based approach [SO00, LZLC02], several model-checking approaches [Esh02, KGMW, Sch99], and an approach to start with EPC’s and a relaxed notion of soundness and to end with WF-nets and soundness [Deh03] are known to us from the literature. Our approach has several advantages over these approaches. First of all, workflow graphs are basically acyclic free-choice WF-nets, whereas our approach can deal with arbitrary WF-nets. Second, the model-checking approaches all assume that the transition system underlying a workflow process is finite, whereas this might not be the case. As a result, we many conclude that our approach has its merits. Our approach has also some disadvantages, especially when compared to Eshuis’ approach [Esh02, EW02], because we totally abstract from data, whereas Eshuis allows for some basic data expressions, and we consider only predefined correctness criteria (soundness, inheritance relations).
CHAPTER 3

Nets and properties

“A lady is nothing very specific. One man's lady is another man's woman; sometimes, one man's lady is another man's wife. Definitions overlap but they almost never coincide.”

Russell Lynes

The previous chapter states that the verification of workflow process definitions has been neglected since workflow management systems were introduced. It also states that we believe that Petri nets can be used to verify workflow process definitions, and, hence, to fill this gap. This chapter presents the Petri-net related definitions and theorems we need. First, it presents a standard class of Petri nets called labeled P/T nets, which we simply call nets in this thesis. Second, it presents workflow nets, which we simply call WF-nets. A WF-net is a net that models a workflow process definition. Third, it presents properties of WF-nets that we use for verifying or diagnosing the soundness of a WF-net in isolation. Fourth and last, it presents properties of WF-nets that we use to decide whether there exists a life-cycle inheritance relation between two sound WF-nets.

3.1 Nets

This section presents the concept of nets. As mentioned before, (labeled P/T) nets are a standard class of Petri nets. According to, for instance, Peterson [Pet81] and Murata [Mur89], Petri nets were originally introduced by Carl Adam Petri [Pet62]. Since then, a lot of research on the theory of Petri nets has been conducted, which has extended the theory and applied it to many kinds of problems. In [GV03], Girault and Valk focus on three applications domains:

• (flexible) manufacturing,
• workflow management, and
• telecommunications,
but they also list other possible application domains, including:

- distributed software systems,
- logistics,
- multi-processor systems,
- software engineering, and
- asynchronous circuits.

Petri nets use a graphical notation that is very easy to understand. On the other hand, Petri nets can be used to analyze concurrent systems in a formal way. As a result of the large amount of research conducted on Petri nets, a lot of analysis techniques are available today.

(Labeled P/T) nets form a standard class of Petri nets. Readers familiar with Petri nets can browse through this chapter to become familiar with the notations used. A more extensive treatment of Petri nets in general can be found in [Pet81, Rei85, Mur89, DE95, RR98a, RR98b]. This chapter is restricted to those properties that are needed to understand the remainder of this thesis.

### 3.1.1 Structure

Figure 3.1 shows an example net. As Figure 3.1 shows, a net contains two kinds of *nodes* (rectangular and circular) and edges with arrowheads. The rectangular nodes are called *transitions*, the circular nodes are called *places*, and the arrow-headed edges are called *arcs*. Note that in Figure 3.1 (and in many figures to come) some arcs coincide, resulting in arrows that seem to have two arrowheads. For example, the arc...
from place p6 to transition t7 coincides with the arc from transition t7 to place p6. Although they seem to be one double-headed arc, they are in fact two single-headed arcs. To every node, a unique identifier (like p1 or t9 in the example net) is associated. For the remainder of this thesis, we assume that the set of possible identifiers is denoted U. To every transition, a label is associated. The label of a transition indicates the action associated with that transition; we will come back to this later in this chapter. For the remainder of this thesis, we assume that the set of possible labels is denoted L. An arc either connects a place to a transition (like the arc from p1 to t1 in the example net), or a transition to a place (like the arc from t3 to p2 in the example net). Arcs directly connecting places and arcs directly connecting transitions are not allowed.

Note that transition t6 is labeled τ. Later on (see Section 3.5), it becomes clear that the label τ ∈ L is reserved for internal, unobservable, actions. If we are not interested in observing the action associated to a transition, like in this case with transition t6, we label it τ.

**Definition 3.1. Net**

Let P, T ⊆ U be finite and non-empty sets such that P ∩ T = ∅, let F ⊆ (P × T) ∪ (T × P), and let l ∈ T → L. The tuple (P, T, F, l) is a net. Set P is the set of places, set T is the set of transitions, set F is the set of arcs, and function l maps every transition to some label.

For the example net (see Figure 3.1), P, T, F, and l are as follows:

\[
P = \{p1, p2, p3, p4, p5, p6, p7\}
\]

\[
T = \{t1, t2, t3, t4, t5, t6, t7, t8\}
\]

\[
F = \{(p1, t1), (p3, t2), (p3, t3), (p4, t4), (p4, t5), (p5, t3), (p5, t7), (p6, t6), (p6, t7), (p7, t6), (p7, t8), (t1, p4), (t1, p5), (t2, p4), (t3, p2), (t4, p3), (t5, p6), (t6, p4), (t6, p7), (t7, p6), (t7, p7), (t8, p5)\}
\]

\[
l = \{(t1, connect), (t2, reconnect), (t3, archive), (t4, disconnect), (t5, order), (t6, τ), (t7, confirm), (t8, ship)\}
\]

In the standard literature, although there is some consensus on the formalization of Petri nets and their properties, differences exist in these formalizations. For this reason, throughout this chapter, we give an overview of a number of existing formalizations to give the reader an idea how these existing formalizations relate to ours.

In [Pet81], Peterson defines the notion of Petri-net graphs. Our definition of nets extends the definition of Petri-net graphs with the labeling function. We need the labeling function to be able to have multiple transitions correspond to the same action (task). In [Rei85], Reisig defines the notion of place/transition nets. Place/transition nets extend
Petri net graphs with place capacities, arc weights, and an initial state. Later on, it becomes clear that a place can contain a number of abstract objects called tokens, and that transitions can remove tokens from certain places and add tokens to certain places. A place capacity determines an upper bound for the number of tokens in the place, whereas a weight of an arc determines how many tokens are removed from or added to a place by a transition. For our purposes, it is sufficient that place capacities are unlimited. We could have incorporated arc weights into our definition of nets, but this would only hinder the readability of many definitions and theorems to come. Later on, it becomes clear that we do use arc weights on some occasions, but we will also show how arc weights can be mapped to unweighted arcs (that is, arcs with weight 1) in these occasions. Thus, to keep matters simple, we assume that all arcs are unweighted. Finally, we transfer the notion of an initial state to systems, which we present later on. Murata [Mur89] and ISO/IEC [ISO00] define a notion of Petri nets that conforms to Reisig’s place/transition nets, but that assumes (like we do) that place capacities are unlimited. Esparza and Silva [ES90] define place/transition nets (also called P/T nets) in a similar way to Murata and ISO/IEC, but without the initial state. Desel and Esparza [DE95] and Desel and Reisig [DR98] introduce a notion of nets that is similar to ours, but that does not contain a labeling function. Our notion of nets is identical to the notion of labeled P/T nets as used by Van der Aalst and Basten in [BA01, AB02].

A node is a called an input node of a second node if some arc connects the node to the second node. Likewise, a node is a called an output node of a second node if some arc connects the second node to the node (that is, if the second node is an input node of the node). The set of input nodes of a node is called its preset; the set of output nodes is called its postset.

**Definition 3.2. Preset, postset**

\[
\text{Let } N = (P, T, F, l) \text{ be a net and let node } n \in P \cup T. \text{ The set } \{n' \in P \cup T | (n', n) \in F\} \text{ is the preset of node } n \text{ in net } N. \text{ Likewise, the set } \{n' \in P \cup T | (n, n') \in F\} \text{ is the postset of node } n \text{ in net } N. \text{ We use } \bullet(N, n) \text{ to denote the preset of node } n \text{ in net } N, \text{ and we use } (N, n)\bullet \text{ to denote the postset of node } n \text{ in net } N. \text{ If no confusion is possible on the net being considered, we may abbreviate these notations to } n^\bullet \text{ and } n\bullet. \]

In the example net, place p1 has no input transitions (hence, \(p1^\bullet = \emptyset\)) and only transition t1 as output transition (\(p1\bullet = \{t1\}\)); place p4 has transitions t1, t2, and t6 as input transitions (\(p4^\bullet = \{t1, t2, t6\}\)) and both transition t4 and t5 as output transitions (\(p4\bullet = \{t4, t5\}\)); and transition t1 has only place p1 as input place (\(t1^\bullet = \{p1\}\)) and both place p4 and p5 as output places (\(t1\bullet = \{p4, p5\}\)).
Almost all standard literature on Petri nets, for instance [Rei85, Mur89, DE95], use the abbreviated notation for the preset and the postset of a node, although they only occasionally mention any precise notation including the net. Later on, it becomes clear that the full notation is useful when discussing two nets simultaneously. Peterson [Pet81] uses $I(n)$ and $O(n)$ instead of $n\bullet$ and $n\circ$, while ISO/IEC [ISO00] uses $Pre(n)$ and $Post(n)$. Van der Aalst and Basten [AB02] use $n^\bullet$ and $n^\circ$ as precise notations, which we consider to be less readable because of the stacking of symbols.

Sometimes, it is useful to take only a part of the net into consideration. For this reason, we introduce the concepts of subnets and partial subnets. Subnets can be obtained by removing some subset of nodes and any arcs connected to some node in this subset. Of course, the labeling function should reflect the possible removal of transitions.

**Definition 3.3. Subnet**

Let $N_1 = (P_1, T_1, F_1, l_1)$ and $N_2 = (P_2, T_2, F_2, l_2)$ be nets. Net $N_2$ is a subnet of net $N_1$ if and only if $P_2 \subseteq P_1$, $T_2 \subseteq T_1$, $F_2 = F_1 \cap ((P_2 \times T_2) \cup (T_2 \times P_2))$, and $l_2 = l_1 \cap (T_2 \times L)$. We use $N_2 \subseteq N_1$ to denote that net $N_2$ is a subnet of net $N_1$.

If we were to remove transition $t_5$ from the example net, we would have to remove the arcs $(p_4, t_5)$ and $(t_5, p_6)$ as well to obtain a subnet.

Except for the labeling function, this definition and this notation is similar to the definition and notation as used by Esparza and Silva [ES90] and Desel and Esparza [DE95].

Partial subnets can be obtained by removing (i) some subset of nodes and any arcs connected to some node in this subset and (ii) any arc. In other words: A partial subnet is a subnet minus some arcs. As with subnets, the labeling function should reflect the possible removal of transitions.

**Definition 3.4. Partial subnet**

Let $N_1 = (P_1, T_1, F_1, l_1)$ and $N_2 = (P_2, T_2, F_2, l_2)$ be nets. Net $N_2$ is a partial subnet of net $N_1$ if and only if $P_2 \subseteq P_1$, $T_2 \subseteq T_1$, $F_2 \subseteq F_1 \cap ((P_2 \times T_2) \cup (T_2 \times P_2))$, and $l_2 = l_1 \cap (T_2 \times L)$.

If we would only remove the arc $(p_4, t_5)$ from the example net, the resulting net would not be a subnet, but it would be a partial subnet.

This definition is similar to the definition as used by Esparza and Silva [ES90].
A net as defined above is only a static structure: it exhibits no (dynamic) behavior whatsoever. To add behavior to a net, we introduce net states and specify how we can move from one state to another.

3.1.2 Net state

A net state is a distribution of a finite number of abstract objects, called tokens, to the places of that net. As mentioned earlier, we assume the token capacity for any place to be unlimited; that is, any place can be assigned any number of tokens (as long as the number is finite). Such a distribution of tokens over the places of a net is called a marking of that net. Figure 3.2 shows a possible marking of the example net: one token, represented graphically by a black dot, is assigned to place p6 and two tokens are assigned to place p4. In this thesis, a marking is represented by a bag or a multiset. First, we introduce the general concept of bags, using the bag concepts from [Bas98] as a starting point. Second, we present markings.

Definition 3.5.
Bag

Let $A$ be some set and let $b : A \rightarrow \mathbb{N}$, where $\mathbb{N}$ is the set of natural numbers $\{0, 1, 2, \ldots\}$. Function $b$ is a bag over $A$ if and only if $b$ assigns only a finite number of elements from $A$ a non-zero value.

Note that a finite set is also a bag, namely the function assigning 1 to every element in the set and 0 otherwise.

We use $\mu A$ to denote the set of all bags over the set $A$. For a bag $b \in \mu A$ and $a \in A$, $b(a)$ denotes the number of occurrences of $a$ in $b$, often called the cardinality of $a$ in $b$. We use brackets to explicitly enu-
Nets

merate a bag and superscripts to denote cardinalities. For example, \([a^2, b^3, c]\) is the bag with two \(a\)'s, three \(b\)'s and one \(c\); the bag \([a^2]^P(a)\] where \(P\) is a predicate on \(A\), contains two elements \(a\) for every \(a\) such that \(P(a)\) holds. The empty bag is denoted \(\varnothing\). The sum of two bags \(b_1\) and \(b_2\), denoted \(b_1 + b_2\), is defined as \([a^\varnothing|a \in A \land n = b_1(a) + b_2(a)]\). The difference of two bags \(b_1\) and \(b_2\), denoted \(b_1 - b_2\), is defined as \([a^\varnothing|a \in A \land n \in (b_1(a) - b_2(a)) \max 0]\). Bag \(b_1\) is a subbag of \(b_2\), denoted \(b_1 \leq b_2\), if and only if for all \(a \in A\): \(b_1(a) \leq b_2(a)\). Except for \(\mu A\), which conforms to the notation as used by ISO/IEC [ISO00], all other bag-related notations conform to the notations as used by Van der Aalst and Basten (who use \(B(A)\) instead of \(\mu A\)) [BA01, AB02].

**Definition 3.6. Marking**

Let \(N = (P, T, F, l)\) be a net and let \(M \in \mu P\). Bag \(M\) is a marking of net \(N\).

Figure 3.2 shows the marking \([p4^2, p6]\) for the example net: two tokens assigned to place \(p_4\) and one to place \(p_6\).

In the standard literature, a marking is usually a function that assigns to every place a natural number [Pet81, Mur89, DE95] (and, thus, a bag), although Reisig [Rei85] extends this function by allowing to assign a special infinity symbol (\(\omega\)). Because we want markings to be finite, we stick to the usual definition.

The combination of a net and an initial state is often called a system [Pet81, Rei85, Mur89, DE95, DR98]. However, using these systems we cannot distinguish successful termination from deadlock, which is necessary when comparing the behaviors of two nets. Therefore, we extend these systems with a notion of a successful terminal state.

**Definition 3.7. System**

Let \(N = (P, T, F, l)\) be a net and let \(I, O \in \mu P\). The tuple \(S = (N, I, O)\) is a system. The marking \(I\) is the system's initial state, whereas the marking \(O\) is its successful terminal state.

Figure 3.3 shows an example system for the example net. For sake of clarity (and to avoid duplication of the entire net), the system’s initial state (marking \([p1]\)) is shown using triangular shapes, and its successful terminal state (\([p2]\)) is shown using rectangular shapes. Note that these shapes do not correspond to tokens, they only visualize the distribution of tokens in both the initial state and the successful terminal state.

A system is also referred to as a marked net [Pet81, DR98, AB02]. Because Reisig [Rei85] and Murata [Mur89] included the initial state into their definition of nets, they had no need for the concept of a system. The disadvantage of doing so is that the distinction between the
static structure of a net, its possible states, and its possible behaviors, is less clear.

3.1.3 Firing rule

Whether in a system a marking can evolve into another marking, is determined by the transitions of the system’s underlying net and the following firing rule. This firing rule states that a transition

- removes a token from every place in its preset and
- adds a token for every place in its postset

provided that the transition is enabled, that is, provided that enough tokens for removal are present in its preset.

**Definition 3.8. Firing rule**

Let \( N = (P, T, F, l) \) be a net, let \( M_1, M_2 \in \mu P \), and let \( t \in T \). In net \( N \) marking \( M_1 \) enables transition \( t \), if and only if \( \bullet t \leq M_1 \). In net \( N \), marking \( M_1 \) evolves to marking \( M_2 \) by firing \( t \), if and only if \( \bullet t \leq M_1 \) and \( M_2 = M_1 - \bullet t + \bullet \). We use \( N \Rightarrow M_1 \Downarrow t \) to denote that in net \( N \) marking \( M_1 \) enables transition \( t \), and we use \( N \Rightarrow M_1 \Downarrow t \stackrel{\circ}{\Rightarrow} M_2 \) to denote that in net \( N \) marking \( M_1 \) evolves to marking \( M_2 \) when transition \( t \) fires. If the net \( N \) is clear from the context, we may omit it and simply write \( M_1 \Downarrow t \) and \( M_1 \Downarrow t \Rightarrow M_2 \).

In the example net, the marking \([p1]\) only enables transition \( t1 \), thus, \([p1][t1]\). If this transition fires, the marking \([p1]\) evolves into marking \([p4, p5]\), thus, \([p1][t1][p4, p5]\). Likewise, the marking \([p4, p5]\) enables the transitions \( t4 \) and \( t5 \). Firing transition \( t5 \) evolves marking

![Figure 3.3. An example system for the example net.](image-url)

The abbreviated notations conform to those used in [Rei85, Mur89], although they do not introduce precise notations \((N \Rightarrow M_1 \{r\}, N \Rightarrow M_1 \{r\} M_2)\) or a notation similar to \(M_1 \{r\} \). Instead of \(M_1 \{r\} M_2\), Peterson [Pet81] uses a next-state function \(\delta(M_1, r) = M_2\), while Desel and Reisig [DR98] use the notation \(M_1 \Rightarrow M_2\). Van der Aalst and Basten [BA01, AB02] use \((N, M_1) \{r\}\) and \((N, M_1) \{r\} (N, M_2)\) as precise notations, which has the disadvantage that we sometimes have to duplicate net \(N\) and that we have no abbreviated notations.

### 3.2 Running example

The attentive reader may have noticed that the example system captures the control-flow behavior of some workflow. Throughout the coming chapters, we use this example workflow as a running example. This section describes this workflow and explains why we do not use the workflow as described in Chapter 1.

A company has been running a web portal for their free software products for some time now. Through this portal, anyone should be able to obtain these products. To access the portal, a client needs to connect to it by specifying his credentials, containing at least a valid e-mail address. If connected, the client can order a free product. If the client orders a product, the company is obliged to send the product by e-mail to the client. However, the company first waits some time because the client might just order additional products. After the company has waited for five minutes or if the size of the ordered products exceeds one megabyte, the company sends the ordered products by one e-mail. The client can disconnect at any time. Figure 3.4 shows a system that captures this workflow.

However, the company has received some comments from clients. Some clients like to have the option to reconnect, because sometimes they disconnect too early. For example, they disconnect before the e-mail has arrived, and only when the e-mail arrives they learn that they have to order an additional product. The advantage of the reconnect is that clients do not have to specify their credentials again. Other clients would like to have some kind of confirmation that an e-mail is going to be send after they ordered their first product.

A novice engineer has been appointed to modify the web portal accordingly. After some time, the engineer came up with an implementation
whose behavior is captured by the system of Figure 3.3. For the remainder of this thesis, the net (system) shown in Figure 3.3 is referred to as the Revised net (system), whereas the net (system) shown in Figure 3.4 is referred to as the Original net (system). Note that, apparently, transition t6 in the Revised net does not correspond to a task (which we want to observe), but to a routing construct (which we do not want to observe).

The reasons why we do not use the requisition process example from Chapter 1 are twofold. First, the requisition process exhibits only sequential behavior, whereas parallel behavior is very interesting from a verification point of view. Second, the requisition process contains more tasks, making the example more difficult to understand. Because of these two reasons, the requisition process is less suited to explain issues at hand.

3.3 WF-nets

This section sketches the concept of WF-nets. WF-nets form a subclass of nets and are used to model workflow process definitions. This subclass was originally introduced by Van der Aalst in [Aal97].

Not every net corresponds to a proper workflow process definition. A WF-net must satisfy several structural properties. First, a WF-net should have a well-defined begin and end. For this reason, we require a WF-net to have a begin place and an end place. A token in the begin place indicates that the workflow process instance has just been cre-
ated. Because no work can be done on the instance before it was created, this place should have no input nodes. A token in the end place indicates that the instance has been completed. Because no work can be done on the instance after it is completed, this place should have no output nodes. In Petri-net terminology, a node without input nodes is called a source node, and a node without output nodes is called a sink node. Thus, the begin place should be a source place, and the end place should be a sink place. Second, there is not much use in having a part of a workflow process definition that does not help forwarding some instance from its begin place to its end place. For this reason, we require that every node is on some directed path (see also Definition 3.12 on page 49) from the begin place to the end place. Note that, for a relation \( R \), we use \( R^* \) to denote its reflexive and transitive closure, that is, (i) for all \( x \): \( (x, x) \in R^* \) and (ii) for all \( x, y, z \): if \( (x, y) \in R^* \) and \( (y, z) \in R \) then \( (x, z) \in R^* \).

**Definition 3.9.**

**WF-net**

Let \( N = (P, T, F, l) \) be a net. Net \( N \) is a WF-net if and only if

1. there exists exactly one \( i \in P \) such that \( \bullet i = \emptyset \);
2. there exists exactly one \( o \in P \) such that \( o\bullet = \emptyset \);
3. for all \( n \in P \cup T \): \( (i, n) \in F^* \) and \( (n, o) \in F^* \).

Both the Original net and the Revised net are WF-nets, both with place \( p_1 \) as begin place and place \( p_2 \) as end place.

In [Aal98a, VBA01, AB02], the designated place identifiers \( i \) and \( o \) are used to identify the source place and the sink place. This is less clear when considering multiple nets simultaneously. Therefore, for a WF-net \( N \), we use \( \bullet N \) to denote its source place and \( N\bullet \) to denote its sink place.

Considering the behavioral correctness of a workflow process definition, we are, as explained in Chapter 2, interested in the behavior of a single case, that is, in the behavior of one particular workflow process instance. Assuming that the workflow process definition is modeled by some WF-net, it is an obvious choice to have the marking with only one token in its begin place as its initial state, and the marking with only one token in its end place as its successful terminal state.

**Definition 3.10.**

**WF-system**

Let \( N = (P, T, F, l) \) be a WF-net. The system \((N, [\bullet N], [N\bullet])\) is its corresponding WF-system.

Figure 3.3 shows the WF-system corresponding to the Revised net, called the Revised system. Likewise, the WF-system corresponding to the Original net is called the Original system.
The name WF-system is also used in [VBA01], be it without a labeling function and without a successful terminal marking. In [Aal98a, AB02], a WF-system without a successful terminal marking is referred to as a marked WF-net.

In the next chapter, we argue that certain relevant WF-net properties can be mapped onto well-known net properties. For this mapping, it is necessary to add a transition connecting its end place to its begin place. The resulting net is called the short-circuited WF-net.

Definition 3.11. Short-circuited WF-net

Let $N = (P, T, F, l)$ be a WF-net, let $t \in U \setminus (P \cup T)$, let $\varphi P = P$, let $\varphi T = T \cup \{t\}$, let $\varphi F = F \cup \{(t, \bullet N), (N\bullet, t)\}$, and let $\varphi l = l \cup \{(t, \tau)\}$. The net $(\varphi P, \varphi T, \varphi F, \varphi l)$ is the short-circuited net of $N$, denoted $\varphi N$.

Figure 3.5 shows the short-circuited net of the Revised net. Note that the short-circuited WF-net is not a WF-net: it has no begin place or end place anymore. Also note that any WF-net is a subnet of its corresponding short-circuited WF-net: We only need to remove the short-circuiting transition and both arcs connected to it to obtain the original WF-net.

In [Aal98a], overlines are used: $\overline{N} = (\overline{P}, \overline{T}, \overline{F})$, and the short-circuiting transition is identified by $\tau^*$. We feel that the symbol $\varphi$ is more prominent than an overline, especially in running texts. Later on (see Section 3.5), it becomes clear that the label $\tau$ is reserved for internal, unobservable, actions. Because we are not interested in observing the short-circuited transition, we label it $\tau$. 

---

FIGURE 3.5. the short-circuited Revised net.
3.4 Net properties

Petri nets are known for the availability of many analysis techniques. This is a great asset in favor of the use of Petri nets for workflow modeling. These analysis techniques can be used to prove qualitative properties (safety, invariance, deadlock, and so on), and to calculate quantitative properties (response times, waiting times, occupation rates and so on). In this thesis, the primary focus is on verifying qualitative properties.

This section presents properties of nets that are used to analyze nets in later chapters. First, we present structural properties, that is, properties that only depend on the net’s static structure. Second, we present behavioral properties, that is, properties that also depend on the current state and the firing rule.

3.4.1 Structural properties

A sequence of nodes in the net that are connected through arcs, is called a path [VBA01]. If all arcs on a path are in the proper direction, the path is called a directed path [VBA01]. Note that these notions differ slightly from [Rei85, ES90, DE95], where a directed path is called a path and any path that is not a directed path is called an undirected path. As a result, an undirected path might not be a path, which is counter-intuitive. Note that, for a relation $R$, we use $R^{-1}$ to denote its inverse, that is, $R^{-1} = \{(y, x) \mid (x, y) \in R\}$.

**Definition 3.12.** (Directed) path

Let $N = (P, T, F, l)$ be a net and let $n_1, \ldots, n_k \in P \cup T$, for some $k \in \mathbb{N}$, be nodes of $N$. The sequence $(n_1, \ldots, n_k)$ is a path if and only if for all $i, 1 \leq i < k$: $(n_i, n_{i+1}) \in F \cup F^{-1}$. It is a directed path if and only if for all $i, 1 \leq i < k$: $(n_i, n_{i+1}) \in F$.

In the Revised net, the sequences $(t7, p5)$ and $(p4, t5, p6, t7, p5)$ are paths, but not directed paths, because the arc between transition $t7$ and place $p5$ is in the opposite direction. The sequences $(p5, t7)$ and $(p4, t5, p6, t6, p7)$ are directed paths.

Our notation $(n_1, \ldots, n_k)$ conforms the notation as used by Peterson [Pet81] and Esparza and Silva [ES90]. However, often (compare [Rei85, Mur89, DE95]), the notation $n_1\ldots n_k$ is used. We prefer to use the notation with the parentheses and commas because we think it is more distinct, especially in running text.

A net is called connected [DE95] if all nodes are connected through paths. It is called strongly connected [DE95] if all nodes are connected through directed paths.
Definition 3.13. (Strongly) connected net
Let \( N = (P, T, F, l) \) be a net. Net \( N \) is connected if and only if for all \( n_1, n_k \in P \cup T \): there exists a path \( (n_1, \ldots, n_k) \). It is strongly connected if and only for all \( n_1, n_k \in P \cup T \): there exists a directed path \( (n_1, \ldots, n_k) \).

The Revised net is connected, but not strongly connected. For example, there is no directed path from place \( p_2 \) to place \( p_1 \). However, the short-circuited Revised net, shown in Figure 3.5, is strongly connected.

A (directed) path is called elementary [ES90] if it contains every node at most once.

Definition 3.14. Elementary (directed) path
Let \( s = (n_1, \ldots, n_k) \) be a (directed) path. Path \( s \) is elementary if and only if for all \( i, j \): \( 1 \leq i < j \leq k : n_i \neq n_j \).

In the Revised net, the directed path \( (p_4, t_5, p_6, t_7, p_7, t_6) \) is elementary, but the directed path \( (p_4, t_5, p_6, t_7, p_6, t_6) \) is not. Note that an elementary path is by definition a partial subnet, but that it need not be a subnet.

Suppose we have two elementary directed paths that share only their first and their last node. Esparza and Silva [ES90] call any of these paths a handle of the other, while Verbeek, Basten, and Van der Aalst [VBA01] call the combination of the first and last node a handle. In this thesis we use the latter notion of handles.

Definition 3.15. Handle
Let \( N = (P, T, F, l) \) be a net and let nodes \( n_1, n_2 \in P \cup T \). The pair \( (n_1, n_2) \) is a handle in \( N \) if and only if there exist two elementary directed paths from \( n_1 \) to \( n_2 \) sharing only \( n_1 \) and \( n_2 \).

From a verification point of view, we are especially interested in handles concerning a place and a transition. See for instance Figure 3.6, which shows a partial subnet constructed by taking two elementary directed paths that share only place \( p_4 \) and transition \( t_3 \). Suppose we were to start with a token in place \( p_4 \). In the given partial subnet, that token can either follow one elementary directed path or the other, but not both. As a result, in the given partial subnet, the transition \( t_3 \) cannot be enabled, which could be regarded as an anomaly. Something similar holds for the partial subnet that is constructed by two elementary directed paths that share only transition \( t_1 \) and place \( p_5 \), which is shown in Figure 3.7. In this case, both elementary directed paths can be followed in the partial subnet, leading eventually to two tokens in the place \( p_5 \), which could be regarded as an anomaly as well.

Definition 3.16. PT-handle, TP-handle
Let \( N = (P, T, F, l) \) be a net, let \( p \in P \), and let \( t \in T \). The node pair \( (p, t) \) is called a PT-handle in \( N \) if and only if it is a handle in \( N \); the
A node pair \((t, p)\) is called a TP-handle in \(N\) if and only if it is a handle in \(N\).

Figures 3.6 and 3.7 show that the Revised net contains both PT-handles and TP-handles.

The following net properties are relevant, because, as we will see later, some combinations of these properties invalidate soundness. Thus, by examining these properties we might be able to learn why a net is not sound.

A net is called well-handled if it contains no PT-handles and no TP-handles.
Definition 3.17. Well-handled net

Let \( N = (P, T, F, l) \) be a net. Net \( N \) is well-handled if and only if it contains no PT-handles and no TP-handles.

A WF-net is called well-structured if its short-circuited net is well-handled. Note that because, in general, we cannot short-circuit an arbitrary net, we cannot define well-structuredness on arbitrary nets.

Definition 3.18. Well-structured WF-net

Let \( N = (P, T, F, l) \) be a WF-net. WF-net \( N \) is well-structured if and only if net \( \phi N \) is well-handled.

Recall that for a WF-net \( N \), we use \( \phi N \) to denote the short-circuited WF-net.

Because the Revised net contains PT- and TP-handles, it is not well-structured. However, the Original net is well-structured, which follows in a straightforward way from the fact that every transition in this net has exactly one input place and one output place.

A net is called free-choice [DE95] if every two transitions that share some input place, have identical presets. As a result, if some marking enables one of these transitions, it will enable the other one as well. Because both transitions can compete for any token in their input places, the competition is fair (in other words, the choice is free).

According to Peterson [Pet81, p. 207], free-choice nets were originally introduced in [Hac72]. However, the definition of free-choice nets as given by Hack is more restrictive than the definition given here. In fact, the definition we give here is the definition of extended free-choice nets [Mur89, BCD95, DE95]. In this thesis, we will not consider the more restrictive definition. Therefore, we omit the adjective “extended”.

Definition 3.19. Free-choice net

Let \( N = (P, T, F, l) \) be a net. Net \( N \) is free-choice if and only if for all \( t_1, t_2 \in T: \bullet t_1 \cap \bullet t_2 = \emptyset \) or \( \bullet t_1 = \bullet t_2 \).

The Revised net is not free-choice, which is shown in Figure 3.8 (\( \{p5, p6\} \neq \{p6, p7\} \)). The Original net is free-choice, which follows in a straightforward way from the fact that every transition in this net has exactly one input place and one output place.

The class of elementary extended non-self controlling nets extends the class of free-choice nets. A net is called elementary extended non-self controlling [BCD95] if for every two transitions that share some input places, all elementary directed paths from one of these transitions to the other contain some of these shared input places.
Definition 3.20. Elementary extended non-self controlling net

Let $N = (P, T, F, l)$ be a net. Net $N$ is elementary extended non-self controlling if and only if for all $t_1, t_2 \in T$ such that $\bullet t_1 \cap \bullet t_2 \neq \emptyset$: there does not exist an elementary directed path $(t_1, p_1, \ldots, p_n, t_2)$ such that $\bullet t_1 \cap \{p_1, \ldots, p_n\} = \emptyset$.

The Revised net is not elementary extended non-self controlling: Transitions $t_6$ and $t_8$ share only place $p_7$ as an input place, but the elementary directed path $(t_8, p_5, t_7, p_6, t_6)$ does not contain this place. This is shown in Figure 3.9. The Original net is elementary extended non-self controlling, which follows from the fact that this net is free-choice (suppose $\bullet t_1 \cap \bullet t_2 \neq \emptyset$, then $\bullet t_1 = \bullet t_2$, and, thus, $p_n \in \bullet t_1$).

A net is called a state machine [Pet81, Rei85, Mur89] or an $S$-net [DE95] if every transition has exactly one input place and one output place. As a result, in a state machine, the number of tokens never changes. Also, note that a state machine is by definition well-handled, free-choice, and elementary extended non-self controlling.

Definition 3.21. State machine

Let $N = (P, T, F, l)$ be a net. Net $N$ is a state machine if and only if for all $t \in T$: $|\bullet t| = |\bullet t|$ = 1.
The Revised net is not a state machine, because transitions t1, t6, and t7 have two output places and transitions t3, t6, and t7 have two input places. The Original net is a state machine.

A subnet of a net is called an S-component [DE95] of that net if (i) it is a strongly connected state machine and (ii) it contains the preset and postset of every place in that subnet.

**Definition 3.22. S-component**

Let $N_1 = (P_1, T_1, F_1, l_1)$ and $N_2 = (P_2, T_2, F_2, l_2)$ be nets such that $N_2 \subseteq N_1$. Net $N_2$ is an S-component of $N_1$ if and only if $N_2$ is a strongly connected state machine such that for all $p \in P_1$:

\[ \bullet(N_1, p) \cup (N_1, p) \bullet \subseteq T_2. \]

Figure 3.10 shows that the short-circuited Revised net contains two S-components. Note that the Revised net itself contains no S-components, which is a result of the requirement that S-components need to be strongly connected. The short-circuited Original net is a state machine and, thus, an S-component of itself.

A net is called **S-coverable** if each place is covered by some S-component.

**Definition 3.23. S-coverability**

Let $N = (P, T, F, l)$ be a net. Net $N$ is S-coverable if and only if for each place $p \in P$ there is an S-component $(P_1, T_1, F_1, l_1)$ of $N$ such that $p \in P_1$.

The short-circuited Revised net and the short-circuited Original nets are both S-coverable: all places are covered by the S-components.

We conclude this section on structural properties by outlining two well-known [Mur89, DE95, RR98a] invariant properties: place invariants (sometimes called P-invariants or S-invariants) and transition invariants (sometimes called T-invariants). According to Desel and Esparza [DE95], the concepts of these invariants were introduced by Lautenbach in [Lau75].

Recall that in a state machine, the number of tokens in its places is constant. For instance, if we consider the upper S-component shown in Figure 3.10, we could state that the number of tokens in the places $p_1$, $p_2$, $p_3$, $p_4$, and $p_6$ is constant, denoted like follows:

\[ p_1 + p_2 + p_3 + p_4 + p_6 \quad \text{(EQ 3.1)} \]

In the initial state of the Revised system, the only token is assigned to place $p_1$. As a result, in this marking and all markings that can evolve from it, there can be only one token in these places, thus:
Likewise, in the same system, the lower S-component from Figure 3.10 results in the following expression:

\[ p_1 + p_2 + p_3 + p_4 + p_6 = 1 \]  \hspace{1cm} (EQ 3.2)

We could generalize this by assigning to every place a weight. For example, for the Revised system, we could add equations 3.2 and 3.3, yielding the following equation:

\[ p_1 + p_2 + p_5 + p_7 = 1 \]  \hspace{1cm} (EQ 3.3)

**FIGURE 3.10. Both S-components of the short-circuited Revised net.**
Nets and properties

Observe that the weight 2 is assigned to the places p1 and p2, and that the weight 1 is assigned to all other places. Of course, the underlying net of this expression is not a state machine anymore, but we can use expressions like this to analyze the system at hand. Such an assignment of weights to places is called a **place invariant**.

**Definition 3.24. Place invariant**

Let \( N = (P, T, F, l) \) be a net and let \( w \in P \rightarrow \mathbb{N} \). Function \( w \) is a **place invariant** of \( N \) if and only if for all \( t \in T \):

\[
\sum_{p \in \bullet t} w(p) = \sum_{p \in \bullet t} w(p).
\]

Note that we restrict weights to non-negative numbers (\( \mathbb{N} \)), while in the standard literature [Mur89, Rei85, DE95] weights are allowed to be negative. However, in this thesis, we are only interested in invariants containing non-negative weights, which are called **semi-positive** invariants [DE95]. Therefore, we omit the adjective “semi-positive”.

When we swap the roles of places and transitions in the definition, we obtain **transition invariants**. Loosely speaking, a transition invariant states that any marking evolves to itself if all transitions in that invariant would fire.

**Definition 3.25. Transition invariant**

Let \( N = (P, T, F, l) \) be a net and let \( w \in T \rightarrow \mathbb{N} \). Function \( w \) is a **transition invariant** of \( N \) if and only if for all \( p \in P \):

\[
\sum_{t \in \bullet p} w(t) = \sum_{t \in \bullet p} w(t).
\]

Thus, the net effect on the marking if all transitions in a transition invariant fire, is nil. An example of a transition invariant in the Revised net is the following equation:

\[
t_5 + t_6 \quad \text{(EQ 3.5)}
\]

Firing transition \( t_5 \) results in consuming a token from place \( p_4 \) and producing a token for place \( p_6 \). Firing transition \( t_6 \) consumes a token from place \( p_6 \) and produces a token for place \( p_4 \). Thus, the net effect of firing both transitions is nil.

### 3.4.2 Behavioral properties

Behavioral analysis techniques are those techniques that use markings of a net. If they are based on an initial state, these techniques refer to systems instead of to nets.
A basic notion of a system is the set of its occurrence sequences [Mur89]. An occurrence sequence is simply a chain of transition firings.

**Definition 3.26. Occurrence sequence**

Let \( N = (P, T, F, l) \) be a net and let \( t_1, \ldots, t_n \in T \) and \( M_0, \ldots, M_n \in \mu P \), for some \( n \in \mathbb{N} \). The sequence \( s = (M_0, t_1, M_1, \ldots, t_n, M_n) \) is an occurrence sequence that evolves in net \( N \) marking \( M_0 \) into marking \( M_n \) if and only if for all \( i, 1 \leq i \leq n \):

\[
N \Rightarrow M_{i-1}[t_i] M_i.
\]

We use \( N \Rightarrow M_0(s) M_n \) to denote that in net \( N \) the occurrence sequence \( s \) evolves marking \( M_0 \) into marking \( M_n \). When the net \( N \) is clear from the context, we may omit it and simply write \( M_0(s) M_n \).

Note that an empty occurrence sequence (\( n = 0 \)) evolves a marking into itself, that is, for any marking \( M \):

\[
M_0 \Rightarrow M.(M).
\]

Examples of occurrence sequences in the Revised net are

- \([p1], t1, [p4, p5], t4, [p3, p5], t3, [p2])\.
- \([p4, p7], t5, [p6, p7], t6, [p4, p7], t8, [p4, p5])\, and
- \([p4, p5, p6], t7, [p4, p6, p7], t5, [p6^2, p7], t8, [p5, p6^2])\.

Observe that, by definition, if \((M_0, t_1, M_1, \ldots, t_n, M_n)\) is an occurrence sequence, then \((M_i, t_{i+1}, M_{i+1}, \ldots, t_j, M_j)\) is an occurrence sequence too, where \( 0 \leq i \leq j \leq n \).

An occurrence sequence of a system projected onto transitions yields a firing sequence. In [DE95, DR98], firing sequences are called occurrence sequences, while in [Mur89], occurrence sequences are called firing sequences.

**Definition 3.27. Firing sequence**

Let \( N = (P, T, F, l) \) be a net and let \( t_1, \ldots, t_n \in T \), for some \( n \in \mathbb{N} \). The sequence of transitions \((t_1, \ldots, t_n)\) is a firing sequence of net \( N \) if and only if there exist markings \( M_0, \ldots, M_n \in \mu P \) such that \((M_0, t_1, M_1, \ldots, t_n, M_n)\) is an occurrence sequence of net \( N \).

If we remove the markings from the example occurrence sequences mentioned above, we get the firing sequences

- \((t1, t4, t3)\).
- \((t5, t6, t8)\), and
- \((t7, t5, t8)\).

In a net, a marking is called reachable [Pet81, Mur89, DE95] from a second marking if there exists an occurrence sequence which starts with the second marking and ends with the first marking.
Definition 3.28.
Reachability

Let \( N = (P, T, F, l) \) be a net and let \( M_1, M_2 \in \mu P \). In net \( N \), marking \( M_2 \) is reachable from marking \( M_1 \) if and only if \( N \Rightarrow M_1[s]M_2 \) for some occurrence sequence \( s \).

We use \( N \Rightarrow M_1[^*]M_2 \) to denote that in net \( N \) marking \( M_2 \) is reachable from marking \( M_1 \). If the net \( N \) is clear from the context, we may omit it and simply write \( M_1[^*]M_2 \).

In the Revised net, among others, the markings \([p_3, p_5] \), \([p_4, p_5] \), and \([p_2] \) are reachable from the marking \([p_1] \).

The set of all markings that are reachable from a system’s initial state, is called the reachability set [Pet81] of that system.

Definition 3.29.
Reachability set

Let \( N = (P, T, F, l) \) be a net, let \( S = (N, I, O) \), and let \( V \subseteq \mu P \) such that \( V = \{M \in \mu P | [^*]M \} \). Set \( V \) is called the reachability set of \( S \). We use \([S]\) to denote the reachability set of \( S \).

Peterson [Pet81] uses \( R(N, I) \) to denote \([S]\) . Reisig [Rei85] and Desel and Esparza [DE95] use \([I]\) , and Murata [Mur89] uses \( L(N, I) \) (or \( L(I) \) if \( N \) is clear from the context). Our notation conforms to the one used in [Rei85, DE95], except for the fact that we are explicit about the underlying net by using the system instead of only its initial marking.

On the reachability set, we define four important and well-known [Pet81, Mur89, DE95, RR98a] behavioral properties used in this thesis: dead transitions, live transitions, safe places and bounded places.

A transition is **dead** [Pet81, Mur89, DE95] in some system if no reachable marking enables it. In [Pet81, Mur89], a dead transition is also called \( L^0\)-live. A dead transition cannot fire.

Definition 3.30.
Dead transitions

Let \( N = (P, T, F, l) \) be a net, let \( S = (N, I, O) \) be a system, and let \( t \in T \). Transition \( t \) is dead in system \( S \) if and only if there exists no reachable marking \( M \in [S] \) such that \( M[t] \).

In the Revised system, no transition is dead, as can be checked in a straightforward way. However, if the initial state of the system would have been \([p_4, p_5] \) , transition \( t_1 \) would be dead.

A transition is called **live** [Pet81, Mur89, Rei85, DE95] if it can be enabled from every reachable marking. In [Pet81, Mur89], a live transition is also called \( L^4\)-live. A system is called **live** (or \( L^4\)-live) [Pet81, Mur89] if all its transitions are live.
Let $N = (P, T, F, l)$ be a net, let $S = (N, I, O)$, and let $t \in T$. Transition $t$ is live in system $S$ if and only if for all reachable markings $M \in \{S\}$ there exists a reachable marking $M_1 \in \{(N, M, O)\}$ such that $M_1(t)$. System $S$ is live if and only if every transition is live.

In the Revised system, no transition is live: Marking $[p2]$ is reachable and in this marking all transitions are dead. However, the short-circuited Revised system is live.

Note that for liveness it is crucial to know the reachability set of any reachable marking $\{(N, M, O)\}$. For this reason, we need the reachability graph (sometimes called the occurrence graph) [Pet81].

The nodes in a system’s reachability graph are the system’s reachable markings. An edge in this graph from one marking to a second (possibly the same) marking, tells us that in that system a transition is enabled by the first marking and that firing this transition would evolve the first marking into the second. By labeling every edge with the identifier of the firing transition, we are able to tell which transitions are enabled in which reachable marking.

Figure 3.11 shows a graphical representation of the reachability graph of the Revised system. Note that this graph embeds all occurrence sequences that have a reachable marking as its first marking. As a result, in a system a marking is reachable from another marking if there exists a directed path in the system’s reachability graph from the latter marking to the former. Note that we can also use the reachability graph for determining dead transitions: A transition is dead if there exists no edge in the reachability graph labeled with the transition’s identifier.
A place is called safe [Pet81] if it never contains more than one token at the same time. A system is called safe [Pet81, Rei85, Mur89] if all its places are safe.

**Definition 3.33. Safe places, safe systems**

Let \( N = (P, T, F, l) \) be a net, let \( S = (N, I, O) \) be a system, and let \( p \in P \). Place \( p \) is safe in system \( S \) if and only if for all reachable markings \( M \in \{ S \} : M(p) \leq 1 \). System \( S \) is safe if and only if every place is safe.

The Revised system and the short-circuited Revised system are safe. However, consider the partial subnet of the Revised net as shown by Figure 3.12. To obtain the partial subnet, only the arc from place \( p_6 \) to transition \( t_7 \) and the arc from place \( p_7 \) to transition \( t_6 \) are removed. Because the opposite arcs (from transition \( t_7 \) to place \( p_6 \) and from transition \( t_6 \) to place \( p_7 \)) are still there, this difference can easily be missed by an untrained eye. Note that, as a result of removing the arcs, the partial subnet is a WF-net that is not S-coverable. Figure 3.13 shows a part of the reachability graph for the partial subsystem, that is, the system corresponding to this partial subnet (thus, with initial state \( [p_1] \) and successful terminal state \( [p_2] \)). From this part of the reachability graph, we can conclude that the places \( p_2, p_3, p_4, p_5, p_6, \) and \( p_7 \) (that is, all places except place \( p_1 \)) are unsafe in this partial subsystem.

We can generalize the concept of safeness to the concept of \( k \)-safeness (or \( k \)-boundedness). A place is \( k \)-bounded [Pet81] if it will never contain more than \( k \) tokens. A system is \( k \)-bounded [Pet81, Mur89, DE95] if all its places are \( k \)-bounded. A place and a system are bounded [Pet81, Mur89, DE95] if they are \( k \)-bounded, for some \( k \).
Definition 3.34. bounded places, bounded systems

Let $N = (P, T, F, I)$ be a net, let $S = (N, I, O)$ be a system, let $p \in P$, and let $k \in \mathbb{N}$. Place $p$ is $k$-bounded in system $S$ if and only if for all reachable markings $M \in \mathcal{MS}(S)$: $M(p) \leq k$. Place $p$ is bounded if and only if it is $k$-bounded for some $k \in \mathbb{N}$. System $S$ is $k$-bounded if and only if every place is $k$-bounded. System $S$ is bounded if and only if it is $k$-bounded for some $k \in \mathbb{N}$.

Note that a system is safe if and only if it is 1-bounded.

An important feature of bounded systems is that they have a finite reachability set [DE95, Proposition 2.21]. However, for general systems, this is not the case. Consider, for example, the partial subnet as shown in Figure 3.12. Its corresponding system has an infinite reachability set. Figure 3.13 shows that in this system the firing sequence $(t_5, t_6)$ evolves the reachable marking $[p_4, p_5]$ into the marking $[p_4, p_5, p_7]$. To make matter worse, firing that sequence one hundred times would evolve the reachable marking $[p_4, p_5, p_7]$ into the marking $[p_4, p_5, p_7^{100}]$, firing it one million times would evolve it into the marking $[p_4, p_5, p_7^{1000000}]$, and so on. As a result, the reachability set contains an infinite number of states. For this reason, Figure 3.13 shows only a part of the reachability graph.
A solution to cope with an infinite number of states, is the notion of a coverability graph [Mur89] (also called the Karp-Miller graph), which has, according to Finkel [Fin93], its origin in the work of Karp and Miller [KM69]. A coverability graph is a finite variant of a reachability graph. However, we have to pay a price: First, we must extend markings to handle unbounded behavior; second, a system may have a number of possible coverability graphs, whereas it always has one unique reachability graph. First, we present the extended bags; second, we present the notion of a coverability graph.

An extended bag is a bag that can contain an infinite number of elements.

**Definition 3.35. Extended bag**

Let $A$ be some alphabet, let $\omega$ denote infinity, and let $b \in A \mapsto (\mathbb{N} \cup \{\omega\})$. Function $b$ is an extended bag from $A$ if and only if $b$ assigns only a finite number of elements from $A$ a non-zero value.

The set of all extended bags over $A$ is denoted $\omega A$. Using the fact that for all $n \in \mathbb{N}$: $n < \omega \land n + \omega = \omega$, all operations on bags (markings) can be defined for extended bags (markings) in a straightforward way.

An extended bag $M \in \omega P$ is called an extended marking of a net $(P, T, F, l)$. The set of extended markings can be partitioned into a set of finite markings $\mu P$ and a set of infinite markings $\omega P \setminus \mu P$. Note that the markings as defined by Reisig [Rei85] coincide with our notion of extended markings.

A coverability graph of a system is constructed in an incremental way. Initially, it contains only the system's initial state and no edges. Repeatedly, a marking and a transition enabled in that marking are selected such that from the selected marking there exists no outgoing edge labeled with the selected transition. If such a marking and transition do not exist, the coverability graph is constructed completely. Otherwise, we add a new marking, namely the marking resulting from firing the selected transition in the selected marking, to the coverability graph and add an edge from the selected marking to the new marking, labeled with the selected transition. If, in the coverability graph as constructed so far, (i) markings are found that are smaller than the new marking and (ii) from which there exists a directed path to the selected marking (and, thus, there exists a directed path to the new marking after the new marking and the new edge have been added), then the appropriate places in the new marking are mapped to infinity. For the remaining places, the firing rule is used to determine their cardinalities. It is known that a coverability graph is finite [Pet81, Rei85].
Net properties

Definition 3.36. Coverability graph

Let \( N = (P, T, F, l) \) be a net, let \( S = (N, I, O) \) be a system, let \( V \subseteq \omega P \), let \( E \subseteq (V \times T \times V) \) be a set of \( T \)-labeled arcs, and let \( G = (V, E) \) be a graph which can be constructed as follows:

1. Initially, \( V = \{ I \} \) and \( E = \emptyset \).
2. Take an extended marking \( M \in V \) and a \( t \in T \) such that \( M[t] \) and such that no extended marking \( M_1 \) exists with \( (M, t, M_1) \in E \). Let \( M_2 = M_1 - \bullet t + \bullet \). Add \( M_3 \) to \( V \) and \( (M, t, M_3) \) to \( E \), where for every \( p \in P \):
   i. \( M_3(p) = \omega \), if there is a node \( M_1 \in V \) such that \( M_1 \leq M_2 \), \( M_1(p) < M_2(p) \), and there is a directed path from \( M_1 \) to \( M \) in \( G \);
   ii. \( M_3(p) = M_2(p) \), otherwise.

Repeat this step until no new arcs can be added.

Graph \( G \) is called a coverability graph of system \( S \).

The result of this algorithm may vary depending on the order in which markings are considered in the second step [Rei85]. Nevertheless, a coverability graph of a system can be used to analyze the behavior of the system. Also, in case a system is bounded, a coverability graph of this system will be identical to its reachability graph [Pet81, Mur89, RR98a].

Theorem 3.1. Let \( N = (P, T, F, l) \) be a net, let \( S = (N, I, O) \) be a system, and let \( G \) be a coverability graph of system \( S \). If system \( S \) is bounded, then graph \( G \) is identical to the reachability graph of \( S \).

Proof. See [RR98a], Theorem 27, page 147.

Figure 3.14 shows a part of a coverability graph for the partial sub-system of Figure 3.12. The complete coverability graph contains 54 markings and 189 arcs, and is too complex to show. Note that, in contrast, the coverability graph of the Revised net contains only 8 markings and 13 arcs (see Figure 3.11).

Given a system and a coverability graph of this system, every occurrence sequence of the system corresponds to a directed path in the coverability graph. The converse is not necessarily true: There may be directed paths in the coverability graph that do not correspond to any occurrence sequence. Consider, for example, the self-loop for marking \([p_2^\omega, p_3^\omega, p_5^\omega, p_7^\omega]\) labeled \( t_3 \): In the system shown in Figure 3.12 there exists no occurrence sequence where transition \( t_3 \) is continuously enabled, whereas in the coverability graph shown in Figure 3.14 there exists such a directed path. However, a directed path containing only
finite markings does correspond to some occurrence sequence. This conforms to the fact that the coverability graph is identical to the reachability graph if the former contains no infinite markings. The theoretical worst-case complexity of generating a coverability graph is non-primitive recursive space, although for small to medium sized systems (up to 100 transitions) generating a coverability graph is often feasible. See [EN94] for more information on complexity issues.

In [Fin93], Finkel introduced the notion of a minimal coverability graph of a system. A minimal coverability graph of a system with an infinite reachability graph is usually much smaller than a coverability graph of that system. Another advantage is that the minimal coverability graph of a system is unique. However, the minimal coverability graph of a bounded system may differ from its reachability graph.

The set of nodes in the minimal coverability graph is called the minimal coverability set. Basically, the minimal coverability set is the set of maximal markings in the coverability graph. It is known that the minimal coverability set is finite and unique [Fin93, Lemma 3.4].
Definition 3.37. \((Minimal)\) coverability set

Let \( N = (P, T, F, l) \) be a net, let \( S = (N, I, O) \) be a system, and let \( \omega P \) be such that

1. for every finite marking \( M \in \{S\} \) there exists an \( M_1 \in C \) such that \( M \leq M_1 \) and
2. for every infinite marking \( M \in C \setminus \{S\} \) there is an infinite strictly increasing sequence of reachable markings \( (M_0, M_1, M_2, \ldots) \) such that for every place \( p \in P \) and every \( n \geq 0 \): if \( M(p) = \omega \) then \( M_n(p) \geq n \) else \( M_n(p) = M(p) \).

Set \( C \) is a coverability set of \( S \). Set \( C \) is minimal if and only if no proper subset is a coverability set of \( S \).

Given the minimal coverability set and the firing rule, we can construct the minimal coverability graph.

Definition 3.38. Minimal coverability graph

Let \( N = (P, T, F, l) \) be a net, let \( S = (N, I, O) \) be a system, let \( V \subseteq \omega P \) be the minimal coverability set of \( S \), let \( E \subseteq (V \times T \times V) \) be such that \( (M_1, t, M_2) \in E \) if and only if \( N \Rightarrow M_1[r]M_2 \), and let \( G = (V, E) \). Graph \( G \) is called the minimal coverability graph of \( S \).

Figure 3.15 shows the minimal coverability graph for the short-circuited partial subsystem of Figure 3.12. Note that it is much smaller (only 2 markings and 6 arcs) than its coverability graph, compare Figure 3.13. Using a system’s minimal coverability graph, which is usually much smaller than a coverability graph for that system, we can decide whether dead transitions exist, whether the system is safe, and whether it is bounded, but we not whether the system is live.

We end this section on net properties with outlining the soundness property on WF-nets together with a theorem that relates soundness of a WF-net to boundedness and liveness of its short-circuited corresponding system. The behavioral restrictions we impose on a WF-net (and on its corresponding system) can be derived from the soundness requirements introduced in Chapter 2. Recapitulating, a workflow process must always be able to complete a case, completion of a case must always be proper, and every task should contribute to at least one possible execution of the workflow. In a WF-net, completion of a case is signaled by a token in its sink place. Thus, the completion option
means that it must always be possible to produce a token for this place. Proper completion means that, as soon as a token is produced for the sink place, all other places must be empty. The last requirement simply means that a WF-system should not have any dead transitions.

**Definition 3.39. Soundness**

Let \( N = (P, T, F, l) \) be a WF-net. WF-net \( N \) is called sound if and only if for its corresponding system \( S = (N, [\bullet N], [N\bullet]) \) the following conditions hold:

1. For all \( M \in \mathcal{S} \) there exists an \( M_1 \in \mathcal{(N, M, [N\bullet])} \) such that \( [N\bullet] \leq M_1 \).
2. There exists no \( M \in \mathcal{S} \) such that \( [N\bullet] < M \).
3. There exists no \( t \in T \) that is dead in \( S \).

Soundness is originally defined by Van der Aalst [Aal97], where the first condition states that it should always be possible to complete the case properly. However, both definitions are equivalent, since the original, stronger, first condition follows directly from our first two conditions. Furthermore, the original, stronger, first condition resulted in a redundant second condition.

The Revised net is sound, as can be determined from its reachability graph (see Figure 3.11). First, from every reachable marking, the successful terminal state \([p2]\) can be reached. Second, no reachable marking exists that is larger than marking \([p2]\). Third, for every transition there exists some edge in the graph. However, its partial subnet (see Figure 3.12) is not sound, as can be determined from its coverability graph (see Figure 3.14) or from its minimal coverability graph (see Figure 3.15). Both graphs show that the second condition is violated: Markings exist that exceed \([p2]\).

Van der Aalst [Aal97] has proved that soundness corresponds to the well-known properties of boundedness and liveness.

**Theorem 3.2.**

Let \( N = (P, T, F, l) \) be a WF-net. WF-net \( N \) is sound if and only if its short-circuited corresponding WF-system \( (\phi N, [\bullet N], [N\bullet]) \) is bounded and live.

**Proof.**

See [Aal97].
3.5 System equivalence

This section presents properties necessary for deciding whether two systems behave in an equivalent way. Recall that the behavior of a system is captured by its reachability graph. Recall also that we introduced transition labels to indicate which action is associated with a transition. For this reason, we are not interested in the transition identifiers, but instead in their action labels when comparing the behavior of systems. Therefore, we first introduce some variations on existing properties that uses action labels instead of transition identifiers. Second, we state when two labeled reachability graphs exhibit equivalent behavior. For this purpose, we use the well-known branching bisimilarity equivalence relation. Branching bisimilarity is a behavioral equivalence that equates systems with the same (externally) observable behavior, but possibly different internal behavior. Third, we present how we can handle the situation when only one of the systems exhibits certain observable actions. Fourth and last, we present the concept of life-cycle inheritance, which is an inheritance relation between WF-systems that is relevant for this thesis.

We want to be able to distinguish external (or observable) from internal (or silent) behavior. For this reason, we designate the label $\tau \in L$ for internal actions. Furthermore, we introduce a variation on the firing rule that uses action labels instead of transition identifiers. Let $a \in L$. We use $N \Rightarrow M_1[a]$ to denote that in net $N$ marking $M_1$ enables action $a$, that is some transition $t$ labeled $a$, and we use $N \Rightarrow M_1[a \tau]M_2$ to denote that in net $N$ marking $M_1$ evolves to marking $M_2$ when action $a$ is performed, that is, if some transition $t$ labeled $a$ fires. Furthermore, we write $N \Rightarrow M_1[(a \tau)]M_2$ (note the parentheses) if either (i) $N \Rightarrow M_1[a \tau]M_2$ or (ii) $a = \tau$ and $M_1 = M_2$. We use this notation when comparing the behaviors of two systems: If one of the systems can perform some action $a$ at some point, then (i) the other system should also be able to perform this action, unless (ii) it happens to be an internal action. Finally, we write $N \Rightarrow M_1[\tau^*]M_2$ if in net $N$ the marking $M_2$ can be reached from marking $M_1$ by any number (may be 0) of internal actions. If the net $N$ is clear from the context, we may omit it and simply write $M_1[a \tau]$, $M_1[a \tau]M_2$, $M_1[(a \tau)]M_2$, and $M_1[\tau^*]M_2$.

Next, we present the labeled reachability graph, which is a variation on the reachability graph. The only difference between them is that the labeled reachability graph uses action labels to label its edges, whereas the reachability graph uses transition identifiers.

**Definition 3.40.**

**Labeled reachability graph**

Let $N = (P, T, F, l)$ be a net, let $S = (N, I, O)$, let $V = [S]$, let $E = \{(M_1, a, M_2) \in (V \times L \times V) | M_1[a \tau]M_2\}$, and let $G = (V, E)$. Graph $G$ is called the labeled reachability graph of system $S$. 

---

67
A system that is branching bisimilar [GW89, GW96] to another system, must be able to (i) simulate any observable action of the other system, and (ii) simulate any silent action of the other system that effectively reduces the number of observable actions that can happen next, both possibly after performing any number of silent actions. The definition given below is identical to the definition found in [Bas98], which differs slightly from the original definition. However, it has been shown that both definitions are equivalent [GW96, Bas96].

**Definition 3.41. Branching bisimulation**

Let \( N_1 = (P_1, T_1, F_1, I_1) \) and \( N_2 = (P_2, T_2, F_2, I_2) \) be nets, let \( I_1, O_1 \in \mu P_1 \) and \( I_2, O_2 \in \mu P_2 \), let \( S_1 = (N_1, I_1, O_1) \) and \( S_2 = (N_2, I_2, O_2) \) be systems, let \( G_1 = (V_1, E_1) \) and \( G_2 = (V_2, E_2) \) be the labeled reachability graphs of \( S_1 \) and \( S_2 \), let \( R \subseteq (V_1 \times V_2) \), and let \( a \in L \). Relation \( R \) is a branching bisimulation on \( G_1 \) and \( G_2 \) if and only if

1. if \( (M_1, M_2) \in R \) and \( N_1 \models M_1 \langle a \rangle M_3 \) then there exist \( M_4, M_6 \in V_2 \) such that \( N_2 \models M_2 \langle \tau^* \rangle M_4 \), \( N_2 \models M_4 \langle (a) \rangle M_6 \), \( (M_1, M_4) \in R \), and \( (M_3, M_6) \in R \);

2. if \( (M_1, M_2) \in R \) and \( N_2 \models M_2 \langle a \rangle M_4 \) then there exist \( M_3, M_5 \in V_1 \) such that \( N_1 \models M_1 \langle \tau^* \rangle M_3 \), \( N_1 \models M_3 \langle (a) \rangle M_5 \), \( (M_5, M_2) \in R \), and \( (M_5, M_4) \in R \);

3. if \( (M_1, M_2) \in R \) then \( N_1 \models M_1 \langle \tau^* \rangle O_1 \) and \( (M_2 = O_2) \Rightarrow (N_2 \models M_2 \langle \tau^* \rangle O_2) \) and \( (M_2 = O_2) \Rightarrow (N_1 \models M_1 \langle (a) \rangle O_1) \).

Figure 3.17 shows three typical example graphs for branching bisimulations. There exists no branching bisimulation relation between marking \( M_1 \) and marking \( M_4 \), as the internal action that leads from marking \( M_4 \) to marking \( M_5 \) effectively removes the possibility to perform action \( a \), while from marking \( M_1 \) a marking from which we can only perform action \( b \) is not reachable. Thus, marking \( M_5 \) can not be related to any marking in the leftmost graph. Because marking \( M_5 \)
cannot be related, marking $M_4$ cannot be related as well: if both could be related, then there had to exist a marking $M$ reachable from marking $M_1$ through some number of internal actions such that marking $M$ is related to marking $M_3$, and we just concluded that such a marking $M$ does not exist. In contrast to this, marking $M_1$ can be related to marking $M_8$: Because of the arrow back from marking $M_9$ to marking $M_8$, the option for performing action $a$ is not lost when performing the silent action from the latter to the former marking. As a result, marking $M_1$ can be related to marking $M_9$ as well. For reasons similar to marking $M_1$, marking $M_8$ cannot be related to marking $M_4$.

**Definition 3.42. Branching bisimilarity**

Let $N_1 = (P_1, T_1, F_1, l_1)$ and $N_2 = (P_2, T_2, F_2, l_2)$ be nets, let $I_1, O_1 \in \mu P_1$ and $I_2, O_2 \in \mu P_2$, and let $S_1 = (N_1, I_1, O_1)$ and $S_2 = (N_2, I_2, O_2)$ be systems. Systems $S_1$ and $S_2$ are branching bisimilar if and only if a branching bisimulation relation $R$ exists between their reachability graphs such that $(I_1, I_2) \in R$.

In conformance with [Bas98, AB02, BA01], we use $S_1 \sim_b S_2$ to denote that the systems $S_1$ and $S_2$ are branching bisimilar.

If we want to replace the original Web service by the revised Web service, we need to migrate any running instances from the Original net to the Revised net. Thus, the Revised net should incorporate the entire behavior of the Original net. To check this, we want to check branching bisimilarity on both their reachability graphs. However, because both nets are sound WF-nets, and the Revised net contains additional labels (archive, reconnect, and confirm), both graphs cannot be branching bisimilar. Thus, we first need a way to handle these additional labels before testing branching bisimilarity. Therefore, we state how we can handle observable actions that only occur in one of the nets. First, we can simply remove these actions from the net. This is called *encapsulation* or *blocking*. The concept of encapsulation is well understood in the context of process algebra [BW90, BV95], and was defined on nets in [Bas98, BA01, AB02].

**Definition 3.43. Encapsulation**

Let $N = (P, T, F, l)$ be a net, let $B \subseteq L\setminus\{\tau\}$, and let $N_1 = (P, T_1, F_1, l_1)$ be the subnet of net $N$ such that $T_1 = \{t \in T | l(t) \notin H\}$. Net $N_1$ is the $B$-encapsulated net of $N$. 

![Figure 3.17. Typical example graphs for branching bisimilarity.](image-url)
Conform [Bas98, BA01, AB02], we use $\partial_B(N)$ to denote the $B$-encapsulated subnet of net $N$.

Figure 3.18 shows the Revised net after the label reconnect has been blocked ($B = \{\text{reconnect}\}$). Note, that after encapsulating a WF-net, the resulting net might not be a WF-net anymore.

Second, we can simply ignore the actions that only occur in one net, that is, act as if they were silent actions. This is called abstraction or hiding. Like the concept of encapsulation, the concept of abstraction is well understood in the context of process algebra [BW90, BV95], and was defined on nets in [Bas98, BA01, AB02].

**Definition 3.44. Abstraction**

Let $N = (P, T, F, l)$ be a net, let $H \subseteq L$, and let $l_1 : T \to L$ such that for every $t \in T$: if $l(t) \notin H$ then $l_1(t) = l(t)$ else $l_1(t) = \tau$. Net $(P, T, F, l_1)$ is the $H$-abstracted net of $N$.

Conform [Bas98, AB02, BA01], we use $\tau_H(N)$ to denote the $H$-abstracted net of net $N$.

Figure 3.19 shows the Revised net after the labels archive and confirm have been hidden ($H = \{\text{archive, confirm}\}$).

If we combine the blocking of the label reconnect with the hiding of the labels archive and confirm, we obtain a WF-net that has the same set of labels as the Original net does. Thus, we can test these two nets for branching bisimilarity. This combination of encapsulation, abstraction, and branching bisimilarity is captured by the notion of life-cycle

![FIGURE 3.18. An encapsulated Revised net.](image-url)
System equivalence

Life-cycle inheritance, which we define on sound WF-nets. Life-cycle inheritance was introduced by Van der Aalst and Basten [AB97] and applied by them to the workflow domain [AB02]. Note that, for a function \( f \in D \rightarrow R \) we use \( \text{rng}(f) \) to denote the actual elements from \( R \) that are mapped to by this function, that is, \( \text{rng}(f) = \{ r \in R | \exists d \in D : (d, r) \in f \} \).

**Definition 3.45.**

**Life-cycle inheritance**

Let \( N_1 = (P_1, T_1, F_1, l_1) \) and \( N_2 = (P_2, T_2, F_2, l_2) \) be sound WF-nets, let \( S_1 = (N_1, [\bullet N_1], [N_1 \bullet]) \) and \( S_2 = (N_2, [\bullet N_2], [N_2 \bullet]) \) be their corresponding systems, and let \( X \subseteq L \) be such that \( X = \text{rng}(l_2) \setminus \text{rng}(l_1) \). Net \( N_2 \) is a subclass of net \( N_1 \) under life-cycle inheritance, if and only if there exists an \( H \subseteq X \) such that \( S_1 \sim_b (\tau^{\delta_{X \cap H}(N_2)}, [\bullet N_2], [N_2 \bullet]) \).

Conform [Bas98, AB02, BA01], we use \( N_2 \subseteq_{lc} N_1 \) to denote that net \( N_2 \) is a subclass of net \( N_1 \) under life-cycle inheritance.

Figure 3.20 shows the reachability graph of the Original net (right-hand side) and the reachability graph of the Revised net after blocking and hiding (\( H = \{ \text{archive, confirm} \} \), left-hand side). A dotted line between two markings indicates that those markings are present in a possible branching bisimulation relation. Thus, the markings \([p2]\) and \([p3, p5]\) in the left-hand graph are related to the marking \([p2]\) in the right-hand graph, and marking \([p3, p7]\) is related to marking \([p5]\). However, the marking \([p6, p7]\) cannot be related to any marking on the right-hand side, because from marking \([p6, p7]\) we can perform (performing internal actions too) the action ship continually, while this is impossible form any marking on the right-hand side. As a result,
both systems cannot be branching bisimilar. Later on it becomes clear that there is no subset $H$ of the actions archive, confirm, and reconnect such that the resulting systems are branching bisimilar. As a result, the Revised net is not a subclass under life-cycle inheritance of the Original net, and we might not be able to migrate all running instances from the original Web service to the revised Web service.

3.6 Conclusions

We have formalized the concepts that are pivotal to this thesis: soundness and life-cycle inheritance. Soundness has been defined on a subclass of nets called WF-nets (workflow nets), whereas the life-cycle inheritance relation has been defined on a pair of sound WF-nets. Furthermore, we have introduced a number of properties that are either relevant for deciding

- whether a net is a WF-net,
- whether a WF-net is sound, and
- whether there exists a life-cycle inheritance relation between two sound WF-nets,

or that are relevant for diagnosing

- why a net is not a WF-net and
- why a WF-net is not sound.

Finally, we have presented a theorem that relates the soundness of a WF-net to the well-known properties of boundedness and liveness. In the remainder of this thesis, this theorem is used extensively to decide and diagnose soundness.
The previous chapter introduces WF-nets and the soundness property on WF-nets. It also shows that soundness is related to the well-known properties of liveness and boundedness. This chapter exploits this relation for analysis and diagnosis purposes. First, it relates some structural properties of WF-nets, that can help deciding and diagnosing soundness of a WF-net, to soundness. Second, it relates the option-to-complete and proper-completion requirements to liveness and boundedness. Third, it introduces two classes of error sequences: one related to liveness and the option-to-complete requirement, and one related to boundedness and the proper-completion requirement. These sequences can be used to diagnose errors concerning these two soundness requirements. Fourth, it introduces a process to analyze and diagnose soundness of a given WF-net.

4.1 Structural properties related to soundness

Recall that soundness of a WF-net is equivalent to liveness and boundedness of the short-circuited corresponding WF-system (see Theorem 3.2 on page 66). Because the properties of liveness and boundedness are well-known, this theorem is an interesting result: It shows that for the analysis of WF-nets we can focus on boundedness and liveness of short-circuited WF-systems. Boundedness and liveness have been studied extensively in the Petri-net literature [Hac79, Pet81, Rei85, Mur89, EN94, DE95, RR98a]. Existing results can be tuned to the analysis of WF-nets.

"Genius not only diagnoses the situation but supplies the answers."
Robert Graves
In Section 3.4.1, a number of structural properties have been introduced. Despite the fact that WF-nets need not be free-choice, the free-choice property does play a role in diagnosing WF-nets. Also, PT- and TP-handles, S-components and S-coverability, and place and transition invariants all play an important role. The interpretation of non-free-choice constructs, PT- and TP-handles, and S-components in the workflow domain is explained in Section 4.5. In this section, we present results relating structural techniques to soundness of WF-nets.

First, a free-choice WF-net can only be sound if the short-circuited WF-net is S-coverable. Recall that if \( N \) is a WF-net, we use \( \varphi N \) to denote the short-circuited WF-net.

**Theorem 4.1.** Let \( N = (P, T, F, I) \) be a sound, free-choice WF-net. The short-circuited net \( \varphi N \) is S-coverable.

**Proof.**

This proof uses the fact that a free-choice system that is live and bounded for some initial marking has to be S-coverable ([DE95], Theorem 5.6). In our proofs, there is a subtle difference between a presupposition and an assumption. In the context of a certain theorem, a presupposition is an assumption that can be regarded true (due to the assumptions made by the theorem itself), whereas an assumption could be false. Thus, everything that is derived from presuppositions can be regarded true, while something that is derived from an assumption can only be regarded true if the assumption is true.

1. The net \( N \) is a sound WF-net (presupposition).
2. The system \( (\varphi N, [\bullet N], [N\bullet]) \) is live and bounded (Result 1 and Theorem 3.2).
3. The net \( \varphi N \) is live and bounded for initial marking \( [\bullet N] \) (Result 2).
4. The net \( N \) is a free-choice WF-net (presupposition).
5. The net \( \varphi N \) is free-choice (Result 4, Definition 3.11 (Short-circuited WF-net), and Definition 3.19 (Free-choice net)).
6. The net \( \varphi N \) is S-coverable (Result 3, Result 5, and [DE95]).

This theorem was introduced by Van der Aalst in [Aal98a] (first part of Theorem 2) and also appeared in [Aal00c] (Corollary 3). The class of free-choice nets is especially relevant for soundness, because liveness and boundedness of the short-circuited system can be decided in polynomial time [DE95].

In the analysis of WF-nets, this theorem can be used as follows. If a net is a free-choice WF-net such that the short-circuited net is not S-cover-
Structural properties related to soundness

Figure 4.1 shows a variation on the Revised net of Figure 3.1 on page 38. Note that this net does not have a transition labeled $\tau$, but that it does have two transitions labeled order. Furthermore, its short-circuited net is free-choice, but not S-coverable: Places $p_5$ and $p_6$ are not in any S-component, which is explained below. Note that the subnet that is obtained if these places are removed, is a state machine. In fact, this state machine is an S-component.

When trying to find an S-component covering place $p_5$, we have to include transitions $t_4$ and $t_6$ (because they are input transitions for place $p_5$). Place $p_4$ is the only input place for transition $t_4$, therefore, we have to include place $p_4$ as well. As a result of adding place $p_4$, we have to add transitions $t_1$, $t_2$, $t_5$, and $t_7$ as well. Because place $p_8$ is the only input transition of transition $t_7$, we have to add this place as well. However, we are not allowed to do so because transition $t_6$ would have two output places: place $p_5$ and place $p_8$. Thus, there exists no S-component covering place $p_5$. When trying to find an S-component covering place $p_6$, we have to add transitions $t_1$, $t_2$, and $t_6$. Transition $t_6$ has place $p_5$ and place $p_8$ output places, but we already know that there exists no S-component covering place $p_5$. Therefore, we have to add place $p_8$, and, thus, transitions $t_7$ and $t_8$. As a result of adding transition $t_7$, we have to add place $p_4$, and, thus, transitions $t_1$ and $t_2$. However, we are not allowed to add these transitions, as both would have place $p_4$ and place $p_6$ as output places. Thus, there also exists no S-component covering place $p_6$. To solve these problems, the workflow
designer could try to avoid that place $p_4$ has to be included in the same S-component as place $p_5$. For instance, s/he could add an arc from place $p_5$ to transition $t_4$. As a result, place $p_4$ is not the only input place of transition $t_4$ anymore. Indeed, after this arc has been added, the places $p_1$, $p_2$, $p_5$, and $p_6$ are in one S-component and the short-circuited net is S-coverable (but not free-choice). Figure 4.2 shows the free-choice Revised net of Figure 4.1 after the arc has been added. We refer to this net as to the S-coverable Revised net, although, strictly speaking, this net is not S-coverable (its short-circuited net is S-coverable, but the net itself is not).

Second, a WF-net can only be well-structured if its short-circuited net is S-coverable.

**Theorem 4.2.** Let $N = (P, T, F, l)$ be a well-structured WF-net. The short-circuited net $\varphi N$ is S-coverable.

**Proof.** This proof uses the fact that a strongly connected, well-handled net needs to be S-coverable ([ES90], Theorem 3.2).

1. The net $N$ is a WF-net (presupposition).
2. The net $\varphi N$ is strongly connected (Result 1, Definition 3.9 (WF-net), Definition 3.11 (Short-circuited WF-net), and Definition 3.13 ((Strongly) connected net)).
3. The net $N$ is well-structured (presupposition).
4. The net $\varphi N$ is well-handled (Result 3 and Definition 3.18 (Well-structured WF-net)).
The net $\varphi N$ contains no PT-handles and no TP-handles (Result 4 and Definition 3.17 (Well-handled net)).

6. The net $\varphi N$ is S-coverable (Result 2, Result 5, and [ES90, Theorem 3.2]).

This theorem seems similar to the ones presented in [Aal98a] (second part of Theorem 2) and in [Aal00c] (Corollary 4). However, both the second part of Theorem 2 and Corollary 4 are only valid for sound WF-nets, whereas Theorem 4.2 is valid for all WF-nets. As a side remark, note that for a given well-structured WF-net, it can be decided in polynomial time whether or not it is sound [Aal00c] (Corollary 2). Also note that the classes of free-choice WF-nets and well-structured WF-nets are incomparable, that is, there are free-choice nets that are not well-structured and vice versa.

S-coverability of a short-circuited WF-net is a sufficient (but not necessary) condition for safeness and, hence, boundedness of the corresponding system.

**Theorem 4.3.**

Let $N = (P, T, F, l)$ be a WF-net such that the short-circuited net $\varphi N$ is S-coverable. The system $\varphi S = (\varphi N, \bullet N, [N\bullet])$ is safe.

**Proof.**

This proof uses the fact that in an S-component the number of tokens does not change, that the initial state is a marking containing only one token, and that the remainder of the net can only restrict the behavior of an S-component.

1. The net $\varphi N$ is S-coverable (presupposition).
2. Let net $N_1 = (P_1, T_1, F_1, l_1)$ be an arbitrary S-component of net $\varphi N$ and let system $S_1$ be $(N_1, [\bullet N], [N\bullet])$ if $\bullet N \in P_1$ and $(N_1, \emptyset, \emptyset)$ otherwise (Definition 3.10 (WF-system)).
3. For all transitions $t \in T_1$: $[\bullet (N_1, t)] = [(N_1, t)\bullet] = 1$, thus, the numbers of tokens in system $S_1$ does not change when firing a transition (Result 2, Definition 3.22 (S-component), and Definition 3.21 (State machine)).
4. The system $S_1$ has either $[\bullet N]$ or $\emptyset$ as initial state (Result 2).
5. For all markings $M \in [S_1]$: $[M] \leq 1$ (Result 3 and Result 4).
6. The system $S_1$ is safe (Result 5 and Definition 3.33 (Safe places, safe systems)).
7. The places in $P_1$ are safe in system $\varphi S$ (Result 6, Definition 3.22 (S-component): only transitions in subnet $N_1$ interface with the remainder of net $\varphi N$, thus, the remainder of system $\varphi S$ can only restrict the behavior of system $S_1$).
Deciding soundness

8. System $\varphi S$ is safe (Result 7, Result 2, Result 1: all places are covered by some S-component).

Note that a consequence of Theorem 4.3 is that both sound free-choice WF-nets and (sound) well-structured WF-nets have safe short-circuited corresponding systems.

Figure 4.3 shows the reachability graph of the short-circuited WF-system corresponding to the S-coverable Revised WF-net of Figure 4.2, assuming the short-circuited transition is called $t$. This reachability graph shows that all places are safe. Recall that one S-component contains the places $p_1$, $p_2$, $p_3$, $p_4$, $p_7$, and $p_8$, while the other S-component contains the places $p_1$, $p_2$, $p_5$, and $p_6$. Observe that for every S-component there is exactly one corresponding token in every reachable marking.

It is also well-known that place invariants can be used to formulate a sufficient condition for boundedness. A place occurring with a positive weight in a place invariant is said to be covered by that invariant. Observe that a place invariant is a generalization of an S-component, as every S-component induces a place invariant [DE95].

**Theorem 4.4.**

Let $N = (P, T, F, l)$ be a WF-net such that every place $p \in P$ is covered by a place invariant such that the weights of the places $\bullet N$ and $N\bullet$ are identical in every place invariant. The system $\varphi S = (\varphi N, [\bullet N], [N\bullet])$ is bounded.

**Proof.**

This proof uses the fact that nets that are coverable by place invariants are bounded [DE95] (Theorem 2.31) and that short-circuiting a net does not have an effect on a place invariant if the source place and sink place have identical weight.

1. Let $w$ be a place invariant of net $N$ such that $w(\bullet N) = w(N\bullet)$ (assumption).
2. The assignment \( \omega \) is a place invariant of the net \( \varphi N \) (Result 1, Definition 3.11 (Short-circuited WF-net), and Definition 3.24 (Place invariant)).

3. Every place in net \( N \) is covered by a place invariant (presupposition).

4. Every place in the net \( \varphi N \) is covered by a place invariant (Result 1, Result 2, and Result 3).

5. The system \( \varphi S \) is bounded (Result 4 and [DE95]).

Places not covered by such place invariants of a short-circuited WF-net may indicate errors.

4.2 Liveness and boundedness vs. soundness

In this section, we investigate the relation between the soundness of a WF-net and the liveness and boundedness of the corresponding WF-system. First, we show that the proper-completion requirement can be related to boundedness. Second, we show that the option-to-complete requirement can be related to liveness. Third and last, we show that, assuming that the proper-completion requirement is met, the no-dead-tasks requirement can be related to the absence of dead transitions in the short-circuited net.

4.2.1 Proper completion

As the following result shows, an unbounded place in a short-circuited WF-net may be a sign of improper completion.

**Theorem 4.5.** Let \( N = (P, T, F, l) \) be a WF-net for which the proper-completion requirement of Definition 3.39 (Soundness) does not hold. The system \( \varphi S = (\varphi N, [\bullet N], [N\bullet]) \) is unbounded.

**Proof.**

This proof uses the fact that if a WF-net has improper completion, then a marking is reachable that is larger than the successful terminal state. As a result, in the short-circuited system, a marking is reachable that is larger than the initial marking, which results in unboundedness.

1. The net \( N \) exhibits improper completion (presupposition).

2. There exists a marking \( M \in [S] \) such that \( M > [N\bullet] \) (Result 1 and Definition 3.39 (Soundness)).

3. Let marking \( M \in [S] \) be such that \( M > [N\bullet] \) (Result 2 and deduction).
Deciding soundness

4. The marking \((M - [N\bullet]) + [\bullet N]\) is reachable in \(\varphi S\) and \((M - [N\bullet]) > \emptyset\) (Result 3 and Definition 3.11 (Short-circuited WF-net)).

5. For all \(n \geq 0\) the marking \((n \cdot (M - [N\bullet])) + [\bullet N]\) is reachable in \(\varphi S\) and \((M - [N\bullet]) > \emptyset\) (Result 4, Definition 3.10 (WF-system), and Definition 3.11 (Short-circuited WF-net)).

6. All places in \(M - [N\bullet]\) are unbounded in system \(\varphi S\) and \((M - [N\bullet]) > \emptyset\) (Result 5 and Definition 3.34 (Bounded places, bounded systems)); thus, unbounded places exist in system \(\varphi S\).

7. The system \(\varphi S\) is unbounded (Result 6 and Definition 3.34 (Bounded places, bounded systems)).

\[\square\]

Figure 4.4 shows a coverability graph of the WF-system corresponding to the free-choice Revised WF-net. From the four left-most reachable markings, among others, it is clear that this WF-net does not comply to the proper completion requirement of soundness, and that in the short-circuited WF-system the places \(p_5\) and \(p_6\) are unbounded. Note that in this example these places are unbounded even if the net is not short-circuited. However, assume we remove transition \(t_2\), which effectively would remove all unbounded markings from the coverability graph shown in Figure 4.4. Thus, the system would be bounded. However, because the markings \([p_2, p_5]\) and \([p_2, p_6]\) would still be present in the graph, the net would still be able to complete improperly, and
places p5 and p6 would still be unbounded in the short-circuited system.

4.2.2 Option to complete

Non-live transitions in a short-circuited WF-system are a potential sign that a WF-net does not satisfy the completion option.

Theorem 4.6. \(\text{Let } N = (P, T, F, I) \text{ be a WF-net such that the option-to-complete requirement of Definition 3.39 (Soundness) does not hold. The system } \varphi S = (\varphi N, [\bullet N], [N\bullet]) \text{ is not live.}\)

Proof. This proof uses the fact that if a WF-net does not have the option to complete, then there exists a marking from which the successful terminal state cannot be reached. As a result, in the short-circuited net, such a marking also exists, thus, the short-circuited transition cannot be live.

1. The WF-net \(N\) does not have the option to complete (presupposition).
2. There exists a marking \(M \in (\{N, [\bullet N], [N\bullet]\})\) such that \([N\bullet] \notin (N, M, [N\bullet])\) (Result 1 and Definition 3.39 (Soundness)).
3. There exists a marking \(M \in (\{\varphi N, [\bullet N], [N\bullet]\})\) such that \([N\bullet] \notin (\varphi N, M, [N\bullet])\) (Result 2 and Definition 3.11 (Short-circuited WF-net)).
4. The short-circuiting transition is not live in system \(\varphi S\) (Result 3, Definition 3.11 (Short-circuited WF-net), and Definition 3.31 (Live transitions, live systems)).
5. The system \(\varphi S\) is not live (Result 4 and Definition 3.31 (Live transitions, live systems)).

\[\square\]

Figure 4.5 shows the reachability graph of the S-coverable Revised net. From this reachability graph, we conclude that this net does not always have the option to complete: From the reachable marking
Deciding soundness

[p5, p7], it is impossible to reach any marking containing a token in place p2. Because no transition is enabled in this marking, no transitions can be live. Thus, this system is not live. As a direct result, the short-circuited system is not live. Note that from the coverability graph as shown in Figure 4.4, we can deduce exactly the same for the free-choice Revised net. Apparently, the free-choice Revised net does not comply with either the option-to-complete requirement or the proper-completion requirement.

4.2.3 No dead tasks

The remaining soundness requirement on a WF-net is the absence of dead transitions in the corresponding WF-system. A dead transition in a WF-system corresponds to a task in the workflow that can never be executed. Non-live transitions in the short-circuited WF-system, in particular dead transitions, might be a sign of dead transitions in the original WF-system. The question is how dead transitions in a WF-system relate to dead transitions in the short-circuited WF-system. Observe that any occurrence sequence of a system is also an occurrence sequence of its short-circuited system, but that the converse is not necessarily true. Thus, a transition that is dead in the short-circuited system is also dead in the original system, but a transition that is dead in the original system might not be dead in the short-circuited system. However, if we assume that the completion of the system is always proper, a transition that is dead in the system is also dead in the short-circuited system.

Lemma 4.7. Let \( N = (P, T, F, l) \) be a WF-net such that the proper-completion requirement of Definition 3.39 (Soundness) does hold. The reachability sets of systems \( S = (N, [\bullet N], [N\bullet]) \) and \( \varphi S = ([\varphi N, [\bullet N], [N\bullet]]) \) are identical.

Proof. This proof uses the fact that if the proper completion requirement holds for a WF-net, then, in its corresponding WF-system, the only reachable marking containing the sink place is its successful terminal state. Thus, the reachability graph of its short-circuited system extends the reachability graph of the original system with only an arc from its successful terminal state to its initial state. Thus, the reachability sets are identical.

1. The net \( N \) has proper completion (presupposition).
2. For all markings \( M \in |S| \): \( \neg(M > [N\bullet]) \) (Result 1 and Definition 3.39 (Soundness)).
3. For all markings \( M \in |S| \): if \( S \Rightarrow M[t] \) then \( M = [N\bullet] \), where \( t \) is the short-circuiting transition of system \( S \) (Result 2).
The reachability sets \( \{S\} \) and \( \{\varphi S\} \) are identical (Result 3, Definition 3.11 (Short-circuited WF-net), Definition 3.29 (Reachability set), the short-circuiting transition does not add a new state).

The S-coverable Revised net complies with the proper-completion requirement. The reachability set of its corresponding WF-system is shown by Figure 4.5, whereas the reachability graph of its short-circuited corresponding WF-system is shown by Figure 4.3. The only difference between both figures is that the latter contains an arc from marking \([p2]\) to marking \([p1]\) labeled with the short-circuited transition. Thus, the reachability sets are identical.

**Theorem 4.8.**

Let \( N = (P, T, F, l) \) be a WF-net such that the proper-completion requirement of Definition 3.39 (Soundness) does hold. Transition \( t \in T \) is dead in system \( S = (N, [\bullet N], [N\bullet]) \) if and only if it is dead in system \( \varphi S = (\varphi N, [\bullet N], [N\bullet]) \).

**Proof.**

This proof uses the fact that the reachability graph of the short-circuited system extends the reachability graph of the system with only an arc labeled with the short-circuiting transition. Thus, after abstracting from this short-circuiting transition (note that we require that \( t \in T \) and not that \( t \in \varphi T \)), the graphs are identical.

1. Net \( N \) has proper completion (presupposition).
2. The reachability sets \( \{S\} \) and \( \{\varphi S\} \) are identical (Result 1 and Lemma 4.7).
3. Transition \( t \in T \) is dead in \( S \) if and only if it is dead in \( \varphi S \) (Result 2, Definition 3.11 (Short-circuited WF-net), and Definition 3.30 (Dead transitions)).

**4.3 Behavioral error sequences**

Structural errors in a net modeling a workflow, that is, violations of the WF-net requirements (see Definition 3.9 on page 47), are generally easy to find and to correct. Behavioral errors, that is, violations of the soundness requirements (see Definition 3.39 on page 66), are more difficult to locate and to correct.

The results in the previous section show that the sets of unbounded places in a short-circuited WF net, as well as the lists of non-live and dead transitions, may provide useful information for diagnosing behavioral errors. Unbounded places, non-live transitions, and dead transitions all point to different types of behavioral errors in a WF-net.
Deciding soundness

However, experience with verification of workflow processes has shown that this information is not always sufficient for finding the exact cause of an error. In particular, it might be difficult to diagnose violations of the option-to-complete requirement and the proper-completion requirement of soundness (see Definition 3.39). To overcome this problem, we introduce the concept of behavioral error sequences. The idea behind these sequences is relatively simple: We want to find minimal firing sequences such that no continuation of that sequence leads to an error-free result. Such a firing sequence is required to be minimal in the sense that every prefix can still lead to an error-free result. Thus, one can think of behavioral error sequence as scenarios that capture the essence of errors made in the workflow design. Depending on the kind of error one is interested in, different types of behavioral error sequences can be helpful for diagnosing the design.

In the next two subsections, we introduce two types of behavioral error sequences called non-live sequences and unbounded sequences that are particularly useful for diagnosing liveness-related (option to complete) and boundedness-related (proper completion) behavioral errors.

4.3.1 Non-live sequences

The next theorem shows that, under the assumption that no transition is dead and no place is unbounded, a short-circuited WF-system is live if and only if the original WF-system satisfies the option-to-complete requirement of soundness.

**Theorem 4.9.** Let $N = (P, T, F, l)$ be a WF-net such that the system $S = (N, [\bullet N], [N \bullet])$ contains no dead transitions and such that the system $\varphi S = (\varphi N, [\bullet N], [N \bullet])$ is bounded. System $\varphi S$ is live if and only if for all markings $M \in |S|$: $[N \bullet] \in [(N, M, [N \bullet])].$

**Proof.** This proof contains four parts: A, B, C, and D. The A-part (Results A1 to A3) is a kind of preamble that shows that the reachability sets of the reachability graphs of both systems are identical. Using this preamble, the B-part (Results B1 to B9) proves the if-part of the theorem, whereas the C-part (Results C1 to C4) proves its only-if part. The D-part (Result D1) is the conclusion. The if-part of this proof uses the facts that, in the short-circuited WF-system, if the initial state is always reachable and that no transitions are dead, all transitions must be live. The only-if part uses the fact that liveness and boundedness of the short-circuited systems correspond to soundness of the system itself.

A1. The system $\varphi S$ is bounded (presupposition).

A2. The net $N$ has proper completion (Result A1 and Theorem 4.5).

A3. The reachability sets $|S|$ and $|\varphi S|$ are identical (Result A2 and Lemma 4.7).
Behavioral error sequences

B1. For all markings $M \in [S] : [N\bullet] \in [(N, M, [N\bullet])]$ (assumption).

B2. For all markings $M \in [\varphi S] : [N\bullet] \in [(\varphi N, M, [N\bullet])]$ (Result A3, Result B1, and Definition 3.11 (Short-circuited WF-net)).

B3. For all markings $M \in [\varphi S] : [N\bullet] \in [(\varphi N, M, [N\bullet])]$ (Result B2 and Definition 3.11 (Short-circuited WF-net)).

B4. For all transitions $t \in T$: $t$ is not dead in system $S$ (presupposition).

B5. For all transitions $t \in T$: $t$ is not dead in system $\varphi S$ (Result A2, Result B4, and Theorem 4.8).

B6. For all transitions $t \in \varphi T$: $t$ is not dead in system $\varphi S$ (the short-circuiting transition is not dead in system $\varphi S$ and Result B5).

B7. For all transitions $t \in \varphi T$, there exists a marking $M \in [\varphi S]$ such that $M[t]$ (Result B6 and Definition 3.30 (Dead transitions)).

B8. For all transitions $t \in \varphi T$ and for all markings $M \in [\varphi S]$, there exists a marking $M_1 \in [\varphi S]$ such that $M_1[t]$ (Result B3 (marking $[N\bullet]$ is always reachable), Result B7 (from marking $[N\bullet]$, every transitions can be enabled), Definition 3.10 (WF-system)).

B9. The system $\varphi S$ is live (Result B8 and Definition 3.31 (Live transitions, live systems)).

C1. The system $\varphi S$ is live (assumption).

C2. The system $\varphi S$ is live and bounded (Result A1 and Result C1).

C3. The net $N$ is sound (Result C2 and Theorem 3.2).

C4. For all markings $M \in [S] : [N\bullet] \in [(N, M, [N\bullet])]$ (Result C3 and Definition 3.39 (Soundness)).

D1. The system $\varphi S$ is live if and only if for all markings $M \in [S] : [N\bullet] \in [(N, M, [N\bullet])]$ (Result B1, Result B9, Result C1, and Result C4).

Thus, when diagnosing liveness of the short-circuited WF-system, the reachability of the successful terminal state in the original WF-system is pivotal. Therefore, we define for a WF-net a non-live sequence as a minimal firing sequence of its corresponding WF-system, that ends in a marking from which it is no longer possible to reach the successful terminal state.

Non-live sequences can be computed from the reachability graph of the original WF-system. Note that this reachability graph is finite, because the short-circuited WF-system, and hence the original WF-system, is assumed to be bounded. In terms of the reachability graph of a WF-system, a non-live sequence is a firing sequence corresponding to a path in that system’s reachability graph such that
Deciding soundness

1. the first marking on the path is the initial state,
2. from the last marking on the path there exists no path to the successful terminal state, and
3. from all other markings on the path there exists a path to the successful terminal state.

Apparently, in case of a non-empty non-live sequence, firing the last transition in that sequence removes the option to complete, and, therefore, hazards liveness of the short-circuited WF-system.

To compute non-live sequences, we first partition the reachable markings of the WF-system into three parts:

1. red markings, from which there is no path to the successful terminal state,
2. green markings, from which paths lead to the successful terminal state but from which no paths lead to a red marking, and
3. yellow markings, from which paths lead to the successful terminal state and from which paths lead to a red marking.

For a non-live sequence, the last marking has to be red whereas all other markings have to be yellow. By definition, such other marking cannot be green (because red markings cannot be reached from green markings), while a red other marking would violate the minimality of the sequence. Figure 4.6 shows the essence of this partitioning. Some remarks:

1. If there is a way to complete properly, then the successful terminal state is green.
2. If there is no way to complete properly, then all markings are red.
3. If there are no red markings, there can be no yellow markings.
4. If there are no green markings, there can be no yellow markings.

FIGURE 4.6. Essence of partitioning a reachability graph for non-live sequences.
All we need to do now is to find arcs in the reachability graph that connect a yellow marking to a red marking. The label of such an arc gives us the transition whose firing removes the option to complete. Any elementary path in the reachability graph from the initial state to a red marking, having a yellow marking as last-but-one marking, corresponds to a non-live sequence.

The definition of non-live sequences can be formalized as follows. Note that the definition does not require the absence of dead transitions in the WF-system under consideration. However, note that liveness of the short-circuited WF-system only corresponds to the reachability of the successful terminal state in the original WF-system under the assumption that no transition is dead.

**Definition 4.1. Non-live sequence**

Let $N = (P, T, F, l)$ be a WF-net such that the system $S = (N, \bullet N, [N\bullet])$ is bounded, and let $G = (V, E)$ be the reachability graph of system $S$. The tuple $(V_R, V_G, V_Y)$ is the liveness-related partitioning of set $V$ if and only if

1. $V_R = \{ M \in V | [N\bullet] \subseteq [(N, M, [N\bullet])]\}$,
2. $V_G = \{ M \in V | V_R \cap [(N, M, [N\bullet])] = \emptyset\}$,
3. $V_Y = \nabla (V_R \cup V_G)$. 

Let $(M_0, t_1, M_1, \ldots, t_{n-1}, M_{n-1}, t_n, M_n)$, for some $n \in \mathbb{N}$, be an elementary directed path in graph $G$ such that $M_0 = \bullet N$. Firing sequence $(t_1, \ldots, t_{n-1}, t_n)$ is called non-live if and only if marking $M_n \in V_R$ and for all $0 \leq i < n$ : marking $M_i \in V_Y$.

If the empty firing sequence is non-live, then we know that we cannot reach the successful terminal state from the initial state. In case of a non-empty non-live sequence, the most valuable information is the combination of its last two markings and its last transition. The only interest we have in the sequence’s prefix is that it gives us a path which leads to the last-but-one marking.

Note that we have excluded firing sequences containing cycles (by requiring that all markings in a non-live sequence must be distinct); cycles do not provide any additional useful information. Also note that, because we do not impose any restrictions on the ‘yellow part’ of non-live sequences, it is possible that several non-live sequences share the same suffix. As a result, because there might be many paths leading to a specific yellow marking that can act as last-but-one marking in a non-live sequence, there may be many non-live sequences with that marking as last-but-one marking.

**Theorem 4.10.** Let $N = (P, T, F, l)$ be a WF-net such that the system $S = (N, \bullet N, [N\bullet])$ contains no dead transitions and is bounded.
Deciding soundness

System $\phi S = (\phi N, \bullet N, [N\bullet])$ is live if and only if system $S$ has no non-live sequences.

Proof.

This follows immediately from Theorem 4.9 and Definition 4.1 (Non-live sequence).

Note that, based on Theorem 3.2, Theorem 4.10 can alternatively be formulated as follows. If the WF-system corresponding to a WF-net does not contain dead transitions and if the short-circuited WF-system is bounded, then the WF-net is sound if and only if the WF-system has no non-live sequences.

Figure 4.7 shows the partitioning for the reachability graph of the WF-system corresponding to the S-coverable Revised net of Figure 4.2. Note that the arcs labeled $t_2$ and $t_8$ are dotted, this is explained in the next section. From this reachability graph, we learn that the firing sequence $(t_1, t_5, t_6, t_7, t_5)$ is non-live: From marking $[p_4, p_5]$, we can still complete properly, but after transition $t_5$ has fired from that marking (which results in marking $[p_5, p_7]$), proper completion is not possible anymore. Apparently, the combination of the reachable marking $[p_4, p_5]$ and transition $t_5$ has been erroneously dealt with by the workflow designer. In this particular case, we can achieve liveness (and, hence, also soundness) by restricting the firing of transition $t_5$ to those markings which contain place $p_6$, by adding two arcs: one from place $p_6$ to transition $t_5$ and one from transition $t_5$ to place $p_6$.

4.3.2 Unbounded sequences

Like the option-to-complete requirement is strongly related to liveness, the proper-completion requirement of a WF-net is strongly related to the boundedness of the its short-circuited WF-system, which is confirmed by the following theorem.

---

FIGURE 4.7. Non-liveness partitioning for the S-coverable Revised net.
Behavioral error sequences

Theorem 4.11. Let \( N = (P, T, F, l) \) be a WF-net. System \( \varphi S = (\varphi N, [\bullet N], [N\bullet]) \) is bounded if and only if system \( S = (N, [\bullet N], [N\bullet]) \) is bounded and the proper-completion requirement of Definition 3.39 (Soundness) holds.

Proof. Like Theorem 4.9, the proof of this theorem contains four parts: a preamble (Results A1 and A2), an if-part (Results B1 and B2), an only-if part (Results C1 to C3), and a conclusion (Result D1). The if-part of the proof uses the fact that the reachability sets of both systems are identical. The only-if part uses Theorem 4.5.

A1. The net \( N \) has proper completion (assumption).

A2. The reachability sets \( [S] \) and \( [\varphi S] \) are identical (Result A1 and Lemma 4.7).

B1. The system \( S \) is bounded (assumption).

B2. The system \( \varphi S \) is bounded (Result A2, Result B1, Definition 3.34 (Bounded places, bounded systems)).

C1. The system \( S \) is bounded (assumption).

C2. The system \( S \) is bounded (Result C1, Definition 3.11 (Short-circuited WF-net), Definition 3.34 (Bounded places, bounded systems)).

C3. The net \( N \) has proper completion (Result C2 and Theorem 4.5).

D1. The system \( \varphi S \) is bounded if and only if the system \( S \) is bounded and the net \( N \) has proper completion (Result A1, Result B1, Result B2, Result C1, Result C2, and Result C3).

Thus, when diagnosing unboundedness of a short-circuited WF-system, boundedness of the original WF-system and reachable markings larger than the successful terminal state are pivotal. Therefore, we define for a WF-net an unbounded sequence as a minimal firing sequence of its corresponding WF-system, that ends in a marking that unavoidably leads to either unbounded places or markings that are larger than the successful terminal state.

Because unbounded sequences are only relevant when diagnosing an unbounded short-circuited WF-system, and because the original WF-system might be unbounded as well, we have to take into account that we cannot use the reachability graph of the original WF-system, as this graph might be infinite. Therefore, in general, we have to resign to using a coverability graph of the system for computing unbounded sequences.

Unfortunately, coverability graphs cannot always be used to compute unbounded sequences accurately. For example, Figure 4.8 shows a WF-net and a possible coverability graph for this net. In this coverabil-
Deciding soundness

In the coverability graph, the initial state unavoidably leads to the unboundedness of places p3 and p4. As a result, the empty firing sequence would be unbounded. However, there does exist a continuation of the empty firing sequence that does not lead to either unbounded places or markings that are larger than the successful terminal state: \((t_1, t_4, t_3, t_2)\). (Note that in the coverability graph the proper completion of this firing sequence is masked by the infinite marking \([p^{2\omega}, p^{3\omega}, p^{4\omega}]\).) Thus, the firing sequences \((t_1, t_4, t_3, t_3)\) and \((t_1, t_4, t_3, t_4)\) are unbounded. Nevertheless, although coverability graphs are clearly not always perfect for computing unbounded sequences, we do not really have an alternative. As a result, one should be aware of this discrepancy when using unbounded sequences to diagnose errors in a WF-net.

To compute unbounded sequences, we partition a given coverability graph in a way similar to the partitioning of the reachability graph for computing non-live sequences given in Definition 4.1:

1. the green markings are those markings from which infinite markings or markings larger than the successful terminal state are not reachable,
2. the red markings are those markings from which no green marking is reachable,
3. the yellow markings are those markings from which infinite markings or markings larger than the successful terminal state are reachable and from which green markings are reachable.

Figure 4.9 shows the essence for partitioning a coverability graph for unbounded sequences. Some additional remarks:

1. Reachable infinite markings are red.
2. Reachable markings larger than the successful terminal state are red.
3. If there are no red markings, there can be no yellow markings.
4. If there are no green markings, there can be no yellow markings.

In the following definition, the notation \( [(G, M)] \) is used to denote the set of all nodes in graph \( G \) that are reachable from node \( M \). Thus, if \( N \) is a net and if \( G \) is the reachability graph of net \( N \), then the notations \( [(G, M)] \) and \( [(N, M, [N\bullet])]) \) are equivalent. However, if net \( N \) is unbounded and if \( G \) is a coverability graph of net \( N \), then the notations \( [(G, M)] \) and \( [(N, M, [N\bullet])]) \) may not be equivalent.

**Definition 4.2. Unbounded sequence**

Let \( N = (P, T, F, l) \) be a WF-net, let \( G = (V, E) \) be a coverability graph of system \( S = (N, [\bullet N], [N\bullet]) \), and let \( V^0 = (V \cup P) \cup \{ M \in [S] | M > [N\bullet]\} \), that is, \( V^0 \) contains the problematic markings. The tuple \( (V_G, V_R, V_Y) \) is the boundedness-related partitioning of set \( V \) if and only if

1. \( V_G = \{ M \in V | V^0 \cap [(G, M)] = \emptyset \} \),
2. \( V_R = \{ M \in V | V_G \cap [(G, M)] = \emptyset \} \),
3. \( V_Y = V \setminus (V_G \cup V_R) \).

**Theorem 4.12.** Let \( N = (P, T, F, l) \) be a WF-net. System \( φS = (φN, [\bullet N], [N\bullet]) \) is bounded if and only if system \( S = (N, [\bullet N], [N\bullet]) \) has no unbounded sequences.

**Proof.** This follows immediately from Theorem 4.11 and Definition 4.2 (Unbounded sequence).
Figure 4.10 shows the partitioning of the coverability graph for the WF-system corresponding to the free-choice Revised net. From this partitioning, we conclude that the firing sequences \( t_1, t_4 \) and \( (t_1, t_5, t_6, t_7, t_4) \) are unbounded. Given these sequences, we might conclude that something is wrong with transition \( t_4 \). Note that earlier on, we already concluded that transition \( t_4 \) was suspicious, because places \( p_5 \) and \( p_6 \) were not S-coverable because of it.

4.4 Reduction techniques

The previous sections all assume that we use a coverability graph to determine boundedness and a reachability graph to determine liveness. However, both graphs can become very large, even to the extent that the computation of boundedness and liveness may become intractable. For this reason, we describe some techniques to keep the size of these graphs within limits. Note that we do not pretend the following list to be exhaustive: For this thesis, these reductions are nice-to-have, but not must-have, and some techniques like, for example, (stubborn sets an sleep sets [Val94, God95, Var98], sweep-line methods [CKM01], and binary decision diagrams [BCM+90]) need to be investigated first to determine to which extent they can be applied in our case.
Reduction techniques

Basically, there exist two kinds of approaches: Those operating on nets, and those operating on the graphs.

4.4.1 Liveness and boundedness preserving reduction rules

In [Mur89], six liveness and boundedness preserving reduction rules are presented. These reduction rules are examples of the first kind of approaches, as they operate on the nets instead of on the graphs. Based on the structure of a net, these rules can remove places and/or transitions from the net. As a result of applying these rules to a net, most likely, its coverability graph and/or reachability graph will be decreased in size. Figure 4.11 shows these six reduction rules: Any net fragment shown on the left can be reduced to the fragment on the right.

FIGURE 4.11. Six liveness and boundedness preserving reduction rules.
Deciding soundness

without changing the liveness or boundedness characteristics of the net.

Because soundness is closely related to liveness and boundedness, we can apply these reduction rules successfully to WF-nets when trying to determine soundness. Note that the six rules also preserve the WF-net requirements (one source place, one sink place, and every node on a path from the source to the sink place).

4.4.2 Reduction rules for free-choice nets

In [DE95], three reduction rules are presented for free-choice nets. Like the six liveness and boundedness preserving reduction rules mentioned above, these three reduction rules operate on the net.

The first rule, $\phi_A$, is shown in Figure 4.12. In free-choice nets, a net fragment shown on the left (note that the two crossed arcs may not exist), can be reduced to the net fragment shown on the right. The second rule, $\phi_S$, states that if we can find a number $n > 0$ and places $p_0, \ldots, p_n$ such that the number of tokens in place $p_0$ is nonnegative linearly dependent on the numbers of tokens in places $p_1, \ldots, p_n$, that is, if positive weights $w_0, \ldots, w_n$ exist such that the equation $w_1p_1 + \ldots + w_np_n = w_0p_0$ holds, then we can remove place $p_0$ from the net. In a similar way, the third rule, $\phi_T$, states that if we can find a number $n > 0$ and transitions $t_0, \ldots, t_n$ such that the effect of firing transition $t_0$ is nonnegative linearly dependent on the effect of firing the transitions $t_1, \ldots, t_n$, that is, if positive weights $w_0, \ldots, w_n$ exist such that the equation $w_1t_1 + \ldots + w_nt_n = w_0t_0$ holds, then we can remove transition $t_0$ from the net.

In [DE95] is has been shown that these three rules can reduce a free-choice net to a net $\{\{p\}, \{t\}, \{(t, p), (p, t)\}, \{(t, l)\}\}$, for some $p, t \in U$ and $l \in L$, if and only if the net is live and bounded. As a result, we can use these rules on a short-circuited free-choice WF-net to determine whether the WF-net is sound. Thus, we would not need any coverability graph or reachability graph for free-choice nets if we would use these three reduction rules. However, as mentioned above,
these rules only operate on free-choice net, and are, therefore, not generally applicable.

4.4.3 Restricted coverability graph

A third approach aims to reduce the size of a coverability graph or reachability graph. Recall that unboundedness and unbounded sequences of a WF-system can be determined using a coverability graph of that WF-system. A simple observation alleviates (to some extent) the problem of large coverability graphs: Infinite markings in a coverability graph have only infinite markings as successors.

For determining unboundedness and unbounded sequences, it is not necessary to consider successors of infinite markings, because they are guaranteed to be infinite. This observation leads to the notion of a restricted coverability graph of a system. Consider again the construction algorithm for coverability graphs (see Definition 3.36 on page 63). For the restricted coverability graph, the marking in step 2 of the construction algorithm is required to be finite.

**Definition 4.3. Restricted coverability graph**

Let $N = (P, T, F, I, O)$ be a net, let $S = (N, I, O)$ be a system, let $\omega P \subseteq V \times T \times V$ be a set of $T$-labeled arcs, and let $G = (V, E)$ be a graph which can be constructed as follows:

1. Initially, $V = \{I\}$ and $E = \emptyset$.
2. Take a finite marking $M \in V \cap \mu P$ and a transition $t \in T$ such that $M(t)$ and such that no marking $M_1$ exists with $(M, t, M_1) \in E$. Let marking $M_2 = M - \bullet t + \bullet t$. Add marking $M_3$ to $V$ and arc $(M, t, M_3)$ to $E$, where for every $p \in P$:
   i. $M_3(p) = \omega$, if there is a marking $M_1 \in V$ such that $M_1 \leq M_2$.
   ii. $M_3(p) = M_2(p)$, otherwise.

   Repeat this step until no new arcs can be added.

Graph $G$ is called a restricted coverability graph of system $S$.

Figure 4.13 shows the unbounded partitioning using a restricted coverability graph of the system corresponding to the free-choice Revised net. Similar to the algorithm that constructs a coverability graph (see Definition 3.36 on page 63), the result of this algorithm may vary depending on the order in which markings are considered in the second step (the same example from [Rei85] applies).

It is straightforward to check that using a restricted coverability graph (instead of a regular coverability graph) results in identical unbounded sequences. However, using a restricted coverability graph does not
Deciding soundness

necessarily result in an identical set of unbounded places. Basically, we can partition the set of unbounded places into two sets: A set of places that are unbounded only because other places are unbounded, and a set of other unbounded places. The former set can be regarded as the set of ‘second-hand’ unbounded places, whereas the latter set can be regarded as the set of ‘first-hand’ unbounded places. Using a regular coverability graph would yield both types of unbounded places, whereas using a restricted coverability graph yields only the ‘first-hand’ unbounded places. We consider this a pre rather than a con for using the restricted coverability graph, because by correcting all ‘first-hand’ unbounded places we automatically correct all ‘second-hand’ unbounded places.

4.5 Diagnosis process

We have seen a wide range of analysis techniques for nets in general and WF-nets in particular. The goal of this section is to apply these techniques to the analysis of workflow processes in a logical and meaningful order, and to distill useful diagnostic information from the analysis results in case of errors in the workflow.

Figure 4.14 illustrates the diagnosis process we propose, modeled using Protos [Pal97, Wav02]. Note that this process is in fact a work-
Diagnosis process

The rectangles are the basic tasks in the process, where special symbols are used for the initial and the final task (Steps 1 and 14). The circles are similar to places in nets. They are only included at some relevant points in the workflow (as explained below). Steps 2 through 8 and 10 through 12 are XOR-splits and Step 14 is an XOR-join. Analyzing the model of Figure 4.14 yields that it corresponds to a sound WF-net.

The basis for the diagnosis process in Figure 4.14 is Theorem 3.2 (Soundness vs. liveness and boundedness). Thus, the diagnosis process aims at establishing the soundness of a WF-net by showing that the
Deciding soundness

short-circuited corresponding system is live and bounded. As mentioned earlier, liveness analysis is only feasible for bounded systems. Therefore, we have decided to center the diagnosis process around the following three milestones, which have to be achieved in the order specified. The naming of the milestones is chosen in such a way that it fits with standard workflow terminology [WFM96, AHKB03].

**Workflow Process Definition (WPD).** Does the workflow process definition correspond to a WF-net? This milestone can be achieved by determining only structural properties of the net under consideration.

**Proper WPD.** Is the short-circuited corresponding system bounded? For this milestone, we can use both structural and behavioral properties. However, the structural properties might not give us a definite answer, whereas the behavioral properties will. Still, because the structural properties are (usually) computationally much more efficient than behavioral ones, we use them as much as possible in the diagnosis process before switching to behavioral techniques.

**Sound WPD.** Is the (bounded) short-circuited corresponding system live (thus, is the WF-net sound)? For this milestone, at the moment, we only have behavioral properties at our disposal.

Note that we assume that the workflow designer can abort the diagnosis process at any point. If, for example, Step 3 provides the designer with sufficient diagnostic information to correct an error, there is no need to continue the process to its end.

### 4.5.1 Step 1: Start of diagnosis

The diagnosis process starts by mapping a workflow process definition from some workflow tool onto a WF-net, as is sketched by Figure 2.4 on page 27. In this step, the process definition is mapped onto a net representation, applying the abstractions discussed in Section 2.2. Thus, we only take the control-flow perspective into account.

### 4.5.2 Step 2: Workflow process definition?

In this step, it is verified whether the first milestone is satisfied. The first milestone is included to guarantee that the process definition that is being imported from some workflow tool corresponds to a WF-net. We simply check whether all the requirements of Definition 3.9 are satisfied (one source place, one sink place, and all nodes must be related to both places). If the milestone is not satisfied, the diagnosis process ends and the workflow designer must make a correction to the process definition. In this case, we provide diagnostic information such as, for example, the list of source places, the list of sink places, and the
list of tasks that are not connected to any source place and/or any sink place.

4.5.3 Step 3: Thread of control cover?

From a workflow point of view, we would like to see a case as a set of parallel \textit{threads of control}: Each thread specifies that certain tasks have to be executed in a certain (sequential) order to get a certain piece of work completed. In the Revised net of Figure 3.1 on page 38, we have two such threads:

1. The first thread handles the piece of work associated with the client. After the client has connected, s/he can request for any number of files. After requesting for a file, s/he has to wait some time. After s/he has requested all files s/he wants to have sent by e-mail, s/he can disconnect. After s/he has disconnected, s/he can reconnect if s/he forgot to request some files unless her/his requests have been archived.

2. The second thread handles the piece of work associated with the company. After a client has connected, the company replies as soon as the client requests some file. After some time, the company will ship all files requested to the client. After the client has disconnected, the company can archive the client’s requests.

The idea of threads is reflected by the S-components in the short-circuited WF-net: Every S-component in that short-circuited net corresponds to a logical piece of work in the workflow. (See, for example, Figure 3.10 on page 55 that shows the two S-components for the Revised net.) Recall that an S-component is a strongly connected subnet that is a state machine (see Definition 3.22 on page 54). For each S-component in a system’s net, the total number of tokens in its places is always constant. From the strongly connectedness of S-components and the structure of WF-nets, it follows that an S-component in a short-circuited sound WF-net always contains the short-circuiting transition, the source place, and the sink place. Given the initial state (one token in the source place), every place in an S-component is safe, and, thus, bounded, and the system corresponding to a short-circuited WF-net that is S-coverable, is, thus, safe and bounded (see also Theorem 4.3). In addition, since the source place is an element of all S-components in an S-coverable net, every S-component contains exactly one token in every marking reachable from the initial state. This observation conforms to the intuitive notion of threads of control.

It appears that any WF-net should satisfy the requirement that its short-circuited net is S-coverable. A place that does not belong to a thread of control is a suspicious place, because it cannot be related to a logical piece of work. Although it is possible to construct a sound WF-net with
a short-circuited net that is not S-coverable, the places that are not S-coverable in sound WF-nets typically do not restrict transitions from being enabled and are thus superfluous. Take, for example, the sound WF-net shown in Figure 4.15. After short-circuiting, this WF-net has two S-components (one containing the places \{p1, p4, p5, p7\} and one containing the places \{p1, p3, p6, p7\}), but none of these S-components covers the place p2. Removing place p2 would not have any effect on the behavior of the corresponding WF-system, thus, it can be removed. Note that S-coverability is not a sufficient requirement: It is possible to construct an unsound WF-net with an S-coverable short-circuited net. For example the S-coverable Revised net (see Figure 4.2) is not sound, which is shown clearly by its reachability graph (see Figure 4.3).

The diagnostic information that we provide in this step is the list of S-components of the short-circuited WF net, as well as a list of places not contained in any of these S-components. This information can generally be computed efficiently. If there are no uncovered places, the second milestone of the diagnosis process (Proper WPD) has been achieved (see Theorem 4.3), which means that we can continue with liveness analysis (see Figure 4.14).

4.5.4 Step 4: Confusions and mismatches?

At this point, we know that our workflow process definition is not covered by threads of control; in Petri-net terminology, the short-circuited WF-system is not S-coverable. Based on Theorem 4.1 and Theorem 4.2, we may conclude that the WF-net under consideration cannot be well-structured and should not be free-choice. If it is free-choice, we know that it cannot be sound. However, it is possible to have a sound WF-net that is neither free-choice nor well-structured.
Diagnosis process

For example, the Revised net (see Figure 3.1 on page 38) is sound, but it is not free-choice or well-structured. For some more advanced routing constructs, non-free-choice nets and/or non-well-structured nets are inevitable. Notwithstanding these observations, in many practical workflows, non-free-choiceness or non-well-structuredness are signs of design errors, as explained in some more detail below.

Confusions. The diagnostic information that we provide on the free-choice property is the set of confusions. A confusion is a non-free-choice cluster, where a cluster is a connected component of the subnet that would remain after all arcs from transitions to places would be removed from the net. A cluster is non-free-choice if (and only if) it does not satisfy the free-choice property of Definition 3.19 on page 52. An example of a non-free-choice cluster is shown in Figure 4.16.

Transitions that do not satisfy the free-choice property have different presets that are not disjoint. In a workflow context, this means that two tasks share some but not all preconditions. Usually, tasks that share a precondition start alternative branches: They form an XOR-split. Also, a task that has multiple preconditions usually ends a set of parallel branches: It is an and-join. Note that in any non-free-choice cluster at least one transition has multiple preconditions. A non-free-choice cluster is therefore often a mixture of XOR-splits with AND-joins. The non-free-choice cluster in Figure 4.16 contains four XOR-splits and four AND-joins: every place except place p2 corresponds to an XOR-split and every transition except transition t3 corresponds to an AND-join. The XOR-splits are troubled by the AND-joins, because one alternative may be enabled while the other is not. The AND-joins are troubled by the XOR-splits, because the token of a completed parallel branch may get removed before an AND-join is enabled. If possible,

![Figure 4.16. A non-free-choice cluster in the net of Figure 4.15.](image-url)
the XOR-splits and AND-joins must be separated: The routing of a case should be independent of the order in which tasks are executed.

As explained in Chapter 2, most of the workflow management systems available at the moment abstract from states between tasks, which means that process definitions imported from these workflow systems yield, in principle, free-choice WF-nets. Clearly, the search for confusions is only meaningful for workflow management systems that allow non-free-choice constructs.

**Mismatches.** A good workflow design is characterized by a balance between AND/XOR-splits and AND/XOR-joins. Clearly, two parallel flows initiated by an AND-split should not be joined by an XOR-join. Two alternative flows created via an XOR-split should not be synchronized by an AND-join. From a workflow point of view, the situations as depicted in Figure 4.17 are suspicious.

In the leftmost situation, the AND-split t6 is terminated by the XOR-join p4. As a result, the condition that corresponds to place p4 signals completion for two branches that may start in parallel for a case. Thus, completing one branch implies that completion is signaled for both branches. The condition corresponding to place p4 can even signal completion twice. In a workflow, such a construct may correspond to an error. In Petri-net terminology, this means that usually all places of a WF-net should be safe. Note that this kind of error may lead to unboundedness of the short-circuited system and hence to unsoundness.

In the rightmost situation, the XOR-split p5 is terminated by the AND-join t3. One of the alternative tasks will be executed for a case. How-
Diagnosis process

ever, the task corresponding to transition $t_3$ synchronizes both branches and needs completion for both alternative branches; thus, it will never be executed. Note that this kind of error may lead to a non-live short-circuited system and hence to unsoundness.

Both situations depicted in Figure 4.17 describe a so-called non-well-handled pair: A transition-place or place-transition pair with two disjoint paths leading from one to the other. The leftmost situation describes a TP-handle, the rightmost a PT-handle (see Definition 3.16 on page 50). Recall from Definition 3.18 on page 52 that a WF net is well-structured if the short-circuited net is well-handled (see Definition 3.17 on page 52). Although a non-well-handled pair in the short-circuited net is often a sign of potential errors, a WF-net that is not well-structured can still be sound. For example, the net of Figure 4.15 is sound but not well-structured, as it contains both TP-handles and PT-handles, as is shown by Figure 4.18.

The diagnostic information that we provide is a list of all non-well-handled pairs in the short-circuited net; usually, the subset of non-well-handled pairs fully embedded in the non-short-circuited net (that is, both paths between the two nodes of the pair do not contain the short-circuiting transition) provides the most useful information, because they often correspond to the undesirable AND-XOR and XOR-AND mismatches discussed above.

At this point in the diagnosis method, the designer may decide to continue the diagnosis process, even if it is already known that the workflow process definition cannot be sound (based on Theorem 4.1 and Theorem 4.2, as explained above). By doing so, the designer might obtain additional diagnostic information to correct an error.

FIGURE 4.18. A TP-handle and a PT-handle in the net of Figure 4.15.
4.5.5 Step 5: Uniform invariant cover?

A uniform invariant is a place invariant assigning only weights zero and one. The set of all uniform invariants of a WF-net can in general be computed efficiently, although it requires theoretically in the worst-case exponential space. Such place invariants can provide useful information concerning the proper-completion requirement of a WF-net.

As mentioned before, the Revised net (see Figure 3.1 on page 38) has a place-invariant \( p_1 + p_2 + p_3 + p_4 + p_6 \), which happens to be uniform. Because we know that initially there is one token in place \( p_1 \) and upon completion there is one token in place \( p_2 \), we conclude from this invariant that the places \( p_1, p_3, p_4, \) and \( p_6 \) are empty upon completion. Furthermore, we can deduce from Theorem 4.4 (place-invariants vs. boundedness) that a short-circuited WF-system is bounded if all places are covered by uniform invariants. A place that is not covered by a uniform invariant \textit{might} be unsafe or even unbounded. From a workflow point of view, this means that a condition might be fulfilled more than once at a single point in time, which is often undesirable.

Note that this check is less discriminating than the check for S-coverability (Step 3): Every S-component corresponds to a uniform invariant. Thus, every place belonging to an S-component is covered by a uniform invariant. However, a place that does not belong to any S-component might still be covered by a uniform invariant.

The diagnostic information that we provide in this step is the set of uniform invariants of the short-circuited WF net as well as the places that are not covered by these invariants. If all places are covered, the Proper-WPD milestone has been achieved.

4.5.6 Step 6: Weighted invariant cover?

Another structural technique for deciding boundedness of the short-circuited WF-net is simply the check whether all places in the net are covered by some place invariant (thus allowing weights greater than one when compared to the previous step). Clearly, this check is less discriminating than the check performed in the previous step. Places that are not covered by a place invariant might be unbounded. From a workflow point of view, this means that a condition \textit{might} be fulfilled an arbitrary number of times.

The diagnostic information that we provide in this step is (a representation of) the set of place invariants of the short-circuited WF-net as well as the places that are not covered by these invariants. If all places are covered, the Proper-WPD milestone has been achieved.
Diagnosis process

Note that for some workflow management systems, Step 5 and Step 6 will not provide any new information: If a net is not S-coverable, then, in general, chances are slim that it will be covered by invariants. However, for some workflow management systems in particular, these steps do provide new information. For example, as is explained in Chapter 7, Staffware process definitions typically do not correspond to S-coverable nets, but typically are covered by invariants.

4.5.7 Step 7: No improper conditions?

At this point in the diagnosis process, we have indications that some places of the short-circuited system might be unbounded. Unbounded places are referred to as improper conditions. An improper condition in the short-circuited system always indicates a soundness error (related to improper completion; see also Section 4.2 and Section 4.3.2). To determine improper conditions, we use either the restricted coverability graph (see Definition 4.3) or the minimal coverability graph (see Definition 3.38 on page 65 or [Fin93]) of the short-circuited system. This computation can be time and space consuming, but it turns out that computing this graph is feasible for most practical workflows (up to several hundreds of tasks).

Figure 4.19 shows the minimal coverability graph of the free-choice Revised system of Figure 4.1. From this graph, we learn that the places p5 and p6 are unbounded, and that all other places are safe. Note that the latter agrees with the fact that these places are part of an S-component (see Section 4.1).

The diagnostic information consists of the set of improper conditions. If this set is empty, the Proper-WPD milestone has been achieved.

4.5.8 Step 8: No substates?

A substate of a system is a reachable marking such that there is another reachable marking (a ‘superstate’) that is larger than the first. It is not difficult to see that the existence substates contradicts with the sound-
Deciding soundness

The free-choice Revised net contains substates, which follows in a straightforward way from the coverability graph shown in Figure 4.4. Note that all states that are in the coverability graph but not in the minimal coverability graph (see Figure 4.19), are substates.

The minimal-coverability-graph algorithm that is used for computing improper conditions in the previous step allows the easy detection of substates (see [Fin93]). We do not provide any detailed information about substates because we think knowing whether some marking is a substate does not really help diagnosing an error. Instead, it would help if we would know how we can reach such a substate. This information is provided by step 13.

4.5.9 Step 9: Improper scenarios!

If the set of improper conditions in Step 7 of the diagnosis process is not empty, we know that the short-circuited WF-system is unbounded. In case the set of improper conditions provides insufficient information for diagnosing the error(s), we offer the workflow designer the possibility to compute the unbounded sequences of the WF-system, called improper scenarios.

As explained in Section 4.3.2, unbounded sequences are computed by constructing and partitioning a restricted coverability graph of the WF-
system. Recall that it is not possible to use the minimal coverability graph for this purpose.

It is not difficult to see that unbounded sequences that are
1. permutations of the same set of transitions and
2. end with the same transition

all provide the same diagnostic information. Thus, it suffices to consider only a single sequence of such a set. For this reason, we restrict ourselves to unbounded sequences that are covered by a spanning tree of the restricted coverability graph. A spanning tree of a graph is a connected subgraph in the form of a tree that contains all nodes of the graph. The tree-constraint means that between every two nodes there is exactly one undirected path. A spanning tree of a reduced coverability graph can be constructed in a straightforward way during the construction of the graph.

Some of the arcs in the reduced coverability graph of the free-choice Revised system (see Figure 4.13) are dotted, while others are solid. The spanning tree of the graph contains exactly the solid arcs. By restricting ourselves to the spanning tree of the graph, the reported unbounded sequences are exactly the two sequences as were given in Section 4.3.2: (t1, t4) and (t1, t5, t6, t7, t4).

Since at this point in the diagnosis process we know that the short-circuited system, corresponding to the free-choice Revised net, is unbounded and, hence, that the Proper-WPD milestone cannot be achieved, the workflow designer must make a correction to this workflow process definition and restart the diagnosis process with a corrected process definition.

4.5.10 Step 10: No dead tasks?

At some point during the diagnosis, the Proper-WPD milestone has been achieved, possibly after one or more corrections to the original process definition have been made. It remains to establish the third milestone of the diagnosis process. Recall that this part of the process aims at establishing the liveness of the short-circuited WF-system.

Using the minimal coverability graph of the short-circuited WF-system, we provide the set of dead transitions of this system. Recall that Theorem 4.8 (Dead transitions in bounded short-circuited WF systems) implies that this set is precisely the set of dead transitions of the non-short-circuited system. These transitions correspond to dead tasks in the workflow process. Note that the minimal coverability graph might already be available from Step 7 (No improper conditions?) of the diagnosis process; if this is not the case, the minimal coverability graph
is computed at this point. If the WF-system contains dead tasks, the workflow designer must correct the error(s) and restart the diagnosis process with the new process definition.

4.5.11 Step 11: No non-live tasks?

At this point in the diagnosis process, we know that the short-circuited WF-system is bounded, and that it does not have any dead transitions. We compute the reachability graph of the short-circuited system to determine the set of non-live tasks, and present them to the workflow designer. If all tasks are live, the diagnosis process is complete and successful: It has been shown that the short-circuited WF-system is bounded and live which by Theorem 3.2 implies that the underlying WF-net is sound.

4.5.12 Step 12: Non-live tasks!

At this point in the diagnosis process, we know that the short-circuited WF-system is bounded, that it does not have any dead transitions, but that it is not live. In this case, we compute the set of non-live tasks using the reachability graph of the short-circuited system.

In the system corresponding to the short-circuited S-coverable Revised net (see Figure 4.2), none of the transitions is live, which is made clear by its reachability graph (see Figure 4.3): In the reachable marking [p5, p7] no transition is enabled. Clearly, the fact that all transitions are non-live is useful, but it might not be sufficient to diagnose the error(s).

4.5.13 Step 13: Locking scenarios!

If the result of Step 11 or Step 12 indicates that there are non-live transitions, but if this information is not sufficient for diagnosing the error(s), we provide the option to compute the non-live sequences of the WF-system. Non-live sequences are referred to as locking scenarios (because they generally lead to livelocks and/or deadlocks in the workflow process).

The set of locking scenarios is computed from the reachability graph of the WF-system (see Section 4.3.1) and minimized using a spanning tree of the reachability graph. As in Step 9 (Improper scenarios!) of the process, the reason for restricting the scenarios to the spanning tree, is that non-live sequences being permutations of the same set of transitions, and ending with the same transition, provide the same diagnostic information, and that there may be an large number of non-live sequences.
Diagnosis process

From the reachability graph of the S-coverable Revised net (see Figure 4.2) and its spanning tree (see Figure 4.7), we can report one locking scenario: \((t_1, t_5, t_6, t_7, t_5)\).

4.5.14 Step 14: End of diagnosis

The diagnosis process ends with one of three possible conclusions, namely that the imported process definition does not correspond to a WF-net, that it does correspond to a WF-net but is not sound, or that it corresponds to a sound WF-net.

In case of errors, the process definition must be corrected in the workflow tool being used (see Chapter 2), after which the diagnosis process has to be restarted.

4.5.15 Example: the free-choice Revised net

To explain how the diagnosis process works, we apply it on the free-choice Revised net (see Figure 4.1).

**Iteration 1.** As we have observed earlier, the free-choice Implementation net is a free-choice, non-S-coverable WF-net. Thus, the first milestone is achieved. As a result, after Step 4, we will conclude that the net cannot be sound. At this moment, we try to achieve soundness by making the net S-coverable, by adding an arc from place \(p_5\) to transition \(t_4\), as explained before. This results in the S-coverable Implementation net (see Figure 4.2). Note that we did not need to construct a coverability graph (see Figure 4.4) of the free-choice Revised net.

**Iteration 2.** The S-coverable Implementation net is an S-coverable WF-net; thus, the first two milestones are achieved. From the reachability graph of its corresponding system (see Figure 4.3), we learn that, in the short-circuited corresponding system, none of the transitions is dead (Step 10), but that none of them are live either (Step 11). To obtain more useful diagnostic information, we compute the set of non-live sequences of the corresponding system: \(\{(t_1, t_5, t_6, t_7, t_5)\}\). So, firing transition \(t_5\) from marking \([p_4, p_6]\) appears to be no problem, while firing that transition from marking \([p_4, p_5]\) makes completion impossible. At this point, as suggested in Section 4.3.1, we decide to disable transition \(t_5\) in marking \([p_4, p_5]\) by adding two arcs: one from place \(p_6\) to transition \(t_5\) and one from transition \(t_5\) to place \(p_6\). The resulting net is shown in Figure 4.21.

**Iteration 3.** The updated S-coverable Implementation net is an S-coverable WF-net, implying again that the first two milestones are achieved. The reachability graph of its corresponding system net is showed in Figure 4.22. From this reachability graph, we learn that, in
Deciding soundness

the short-circuited system, none of the transitions is dead, and that all are live. Thus the last milestone is also achieved and the net is sound.

4.6 Case study

For testing the usefulness of the steps of the diagnosis process of Figure 4.14, we used seventeen Protos [Pal97, Wav02] models of a workflow process of a travel agency at a university. These seventeen models were chosen from the work of twenty groups of students that designed Protos models from an informal description of the workflow process (see Appendix C), as part of an assignment for a course on workflow management.
There are two reasons why this case study is particularly useful for evaluating the diagnosis process. First, Protos supports WF-nets as a modeling language. Consequently, all steps in the diagnosis process may in principle provide useful information concerning possible errors in workflow process definitions designed in Protos. Second, the assignment was set up in such a way that the students had to use a wide variety of routing constructs in their models. By evaluating seventeen models of this workflow process, it is almost guaranteed that these models also contain a wide variety of errors.

The input for this case study consisted of workflow process definitions developed by 42 industrial-engineering students of the course Workflow Management & Groupware (1R420, autumn 1998; Eindhoven University of Technology) and 15 computing-science students of the course Workflow Management: Models, Methods, and Tools (25756, autumn 1998; University of Karlsruhe). These students formed 20 groups which independently designed Protos models of the workflow in a travel agency. Fourteen of these groups were groups of students from the Eindhoven University of Technology; the other six were groups of students of the University of Karlsruhe.

From the Eindhoven collection of models, we selected eleven reasonably looking solutions; three models were so poor that analyzing them by means of the diagnosis process was not very meaningful. From the Karlsruhe collection, all models were selected. The number of tasks and other building blocks of the models ranged from 54 to 89. These numbers show that the case study was performed on workflow models of more than reasonable size. A snapshot of a(n unsound) Protos model of the travel-agency workflow is shown in Figure 4.23.

The groups of Eindhoven consisted of industrial engineers, who had only a little prior experience in modeling and no background in formal verification. Verification of workflows was only a minor topic of the course Workflow Management & Groupware (1R420) and the students did not practice with the soundness property. Although the groups were told to simulate the workflow process by hand (play the token game) to test their model, not one of them was able to produce a sound model.

In contrast to the groups of Eindhoven, the groups taking the course Workflow Management: Models, Methods, and Tools (25756) in Karlsruhe consisted of computing-science engineers, who did have a background in modeling and verification. Furthermore, the importance of making a correct workflow was emphasized and analysis techniques for nets were treated in the course.

In the end, the Karlsruhe groups delivered better models than the Eindhoven groups. Of the seventeen models we analyzed using the diagno-
Deciding soundness

sis process, five appeared to be sound from the start, all from Karlsruhe groups.

Table 4.1 shows an overview of our efforts to diagnose the seventeen workflow models. It contains the following information:

- The number of iterations needed to produce a sound workflow process definition.
- Diagnostic information (see below for more details).
- The estimated time it took us to produce a sound workflow process definition.

The case study was performed on a Pentium 200 PC with 128 Mb of RAM running Windows NT 4.0. The tool Woflan (see Chapter 8) was used to compute the relevant net properties.

The numbers in the second row of the heading of Table 4.1 refer to the corresponding steps of the diagnosis process of Figure 4.14. An entry implies that, based on the information provided in that step, a correction was made in the model being diagnosed. In case a correction was made in Step 4, it is specified whether this correction was based on a

FIGURE 4.23. A snapshot of a Protos model of the travel-agency workflow.
confusion (column 4c) or on a mismatch (column 4m). Note that the entries do not specify the order in which corrections were made. For example, when diagnosing the model of group 3, four corrections based on Step 4 were made in the initial model, one correction based on Step 13 was made in the second model, and one correction based on Step 3 was made in the third model, but this does not show in Table 4.1.

The information in Table 4.1 shows that Steps 3, 4, 7, 9, 10, and 13 of the diagnosis process of Figure 4.14 are all used to make one or more corrections. In particular Steps 3, 4, 9, and 13 are used quite often. This does not come as a surprise because the diagnostic information provided in these steps has a clear interpretation in the workflow domain.

Of course, it is also interesting to see which steps are not used. All Protos models considered in the case study corresponded to workflow process definitions. Consequently, no corrections were made in Step 2 of the diagnosis process. However, this step is essential in the process.

<table>
<thead>
<tr>
<th>Group</th>
<th>Iterations</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td>1</td>
<td></td>
<td></td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>30</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>1</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>1</td>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td></td>
<td></td>
<td>1</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td></td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>2</td>
<td>1</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>3</td>
<td></td>
<td></td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>2</td>
<td></td>
<td></td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>2</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>7</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

TABLE 4.1. Overview of the results of the travel-agency case study.
Deciding soundness

because the WPD milestone guarantees that the remainder of the diagnosis process is meaningful. The information in Table 4.1 furthermore shows that Steps 5, 6, 8, 11, and 12 were not used to make corrections. However, in one occasion (Group 11; final model), Step 5 showed that the process definition was proper; interestingly, that process definition was not covered by threads of controls, which is usually the case. Step 11 is simply required in the diagnosis process for showing soundness of a workflow process definition. Nevertheless, the results of the case study show that a list of non-live tasks is generally not sufficient for diagnosing an error; in all relevant cases, locking scenarios (Step 13) were computed to obtain more detailed information. Further experience with the diagnosis process might point out that Steps 11 and 13 can be integrated. For similar reasons, also Step 12, which is simply a variant of Step 11, might be integrated with Step 13. This leaves Steps 6 and 8. These steps are usually only relevant if the process definition is non-safe (see Definition 3.33 on page 60). When using a Petri-net-related design tool like Protos, this is rarely true. However, when using a non-Petri-net-related design tool (like, for example, Staffware or MQSeries Workflow, see Chapter 7), both steps might turn out to be useful and, furthermore, come almost for free after Steps 5 and 7.

Besides the above observations about the usefulness of the steps in the diagnosis process, two other interesting observations can be made. In the informal description of the travel-agency workflow process, a distinction was made between private trips and business trips. At several points in the process, the execution of certain tasks or the order of execution depended on this distinction. Consequently, a workflow process definition of the travel-agency process almost always contains a number of choices (XOR-splits) that must be kept consistent. In several models used for the case study, this consistency was not enforced by the workflow process definition. The type of a trip is a typical example of a piece of control data (see Section 2.2). As mentioned in Section 2.2, in our opinion, one should avoid situations where the logical correctness of a process definition depends on the invariance of a piece of control data. Fortunately, the diagnostic information provided by the diagnosis process made it straightforward to correct these models enforcing the consistency via the process definition.

Another interesting observation is that the industrial-engineering students of Eindhoven did not produce a single correct workflow, whereas the computing-science students of Karlsruhe handed in only one flawed model, which was straightforward to correct. In our opinion, the different background of the students causes this discrepancy. Industrial-engineering students have little background in modeling and verification; computing-science students are trained in both skills. Many designers of workflow processes in practice have also little experience in formal verification. The diagnosis process can be a useful aid in
designing correct workflow processes that helps to prevent a lot of problems caused by the implementation of erroneous workflow processes.

Summarizing the results of the travel-agency case study, the diagnosis process proved to be useful for diagnosing and correcting all the seventeen models in reasonable time and with reasonable effort. The results indicate that the diagnosis process of Figure 4.14 is appropriate for verifying complex workflow processes.

4.7 Conclusions

We have introduced a diagnosis process that uses state-of-the-art Petri-net-based techniques to decide soundness on a WF-net, and to provide relevant diagnostic information in case the answer is negative. The diagnosis process exploits the fact that soundness of a WF-net is closely related to two well-known and well-researched properties of Petri nets: liveness and boundedness [Aal98a]. Case studies have showed that the diagnosis process is a useful assistant when correcting errors in a WF-net.

Furthermore, we have introduced the concept of behavioral error sequences. Basically, a behavioral error sequence is a minimal firing sequence that necessarily violates some desirable property, where minimal means that any prefix of the sequence does not necessarily violate this property. In case of the liveness and boundedness properties, these sequences are called non-live sequences and unbounded sequences. Non-live sequences can be used to diagnose any liveness-related errors, whereas unbounded sequences can be used to diagnose boundedness-related errors.

Given a behavioral error sequence, it is usually straightforward to find an error, as no continuation of the sequence will satisfy the corresponding desirable property. Therefore, we can simply simulate the entire behavioral error sequence, and see why the desirable property is out of reach. Usually, this will give the designer of a workflow process definition a clear indication of an error. Furthermore, the fact that the behavioral error sequence is minimal should provide some useful insights, as the property can be satisfied if the last transition of the sequence is rolled back.

Behavioral error sequences need at least a coverability graph of the workflow process definition. As a result, they are not restricted to the Petri-net area (they can also be used for any system that results in a state space), but they might be hard to compute.
Deciding soundness

The entire diagnosis process, including the behavioral error sequences, can be used to check whether a WF-net meets the following requirements:

- The successful terminal state is always reachable.
- The sink place is only marked in the successful terminal state.
- No dead transitions exist, that is, every transition can be executed.

If the WF-net does not meet any of these requirements, then detailed diagnostic information is given that guides the designer of that WF-net towards correcting it.
The previous chapter shows how we can decide soundness on a single WF-net in an effective and efficient way. This chapter investigates how we can decide whether one sound WF-net (called the potential sub WF-net) is a subclass under life-cycle inheritance of another sound WF-net (called the base WF-net). First, it shows that an exhaustive search algorithm may be prohibitively expensive, because of the potential large reachability graphs and the exponential number of possible ways to block or hide the labels only present in the potential sub WF-net (which are called extending labels henceforth). Second, it introduces the concepts of constraints, and shows that structural properties can be used to compute these constraints. Third, it presents a backtracking algorithm which uses these constraints to exclude many possible ways to block or hide the extending labels. Fourth, it presents a small case study that shows that these constraints can be used to effectively reduce the number of branching bisimilarity checks to prevent prohibitively expensive processing times.

5.1 Exhaustive search algorithm

Assume, for the sake of argument, that we want to decide whether the Revised net (see Figure 3.1 on page 38) is a subclass under life-cycle inheritance of the Original net (see Figure 3.4 on page 46). Then, we have to determine whether a subset of the set of extending labels \{archive, confirm, reconnect\} exists such that, after hiding the extending labels in this subset and blocking the remaining extending labels, the resulting reachability graphs are branching bisimilar (see Definition 3.45 on page 71 for the details). Figure 5.1 shows how an
exhaustive search algorithm might perform. The resulting search tree contains eight \(2^3\) possible subsets to check branching bisimilarity on. As a result, for this relatively simple example, we have to check branching bisimilarity eight times, before concluding that the Revised net is not a subclass under life-cycle inheritance of the Original net.

The reachability graphs of the Revised net and the Original net are only small. However, particularly in the workflow domain, these graphs can be large: Up to millions of states each. Therefore, one such a branching bisimilarity check might already be time-consuming, despite the fact that efficient algorithms exist to check branching bisimilarity on these graphs [GV90]. Unfortunately, for an exhaustive search, we might have to check branching bisimilarity many times. (Worst-case, we need to check branching bisimilarity \(2^\left|X\right|\) times, where \(X\) is the set of extending labels.)

**5.2 Constraints**

The combination of large reachability graphs and the exponential number of possible subsets results in an exhaustive search algorithm that is prohibitively expensive in many cases. To alleviate this problem, we aim to reduce the number of branching bisimilarity checks. Note that we do not aim to improve the branching bisimilarity check, as we think it is efficient already, and we also do not aim to improve the size of the reachability graphs, as reducing their size while preserving branching bisimilarity is extremely hard, if possible at all.

To reduce the number of branching bisimilarity checks, we develop the concept of constraints. Basically, a constraint is a pair of sets of extending labels that indicates that, for a successful branching bisimilarity check, we cannot block all labels in the first set and hide all labels in
another set, that is, we either have to hide a label from the first set or we have to block some label from the second set. For this reason, we refer to the first set as the hide set, and we refer to the second set as the block set. A hide set is called a hide constraint if the associated block set is empty, and a block set is called a block constraint if the associated hide set is empty. Thus, a hide constraint indicates that at least one of its labels must be hidden in order to allow a successful branching bisimilarity check, whereas a block constraint indicates that at least one of its labels must be blocked. If a constraint is violated, that is, if all labels from the hide set are blocked and all labels in the block set are hidden, we know that we cannot achieve branching bisimilarity anymore. Effective constraints of both types can be efficiently generated using structural analysis techniques for nets.

For the remainder of this chapter, we use the following notations:

1. Net $N_b = (P_b, T_b, F_b, l_b)$ is the base WF-net, and system $S_b = ([N_b], [N_b, \bullet])$ is the base WF-system.
2. Net $N_s = (P_s, T_s, F_s, l_s)$ is the potential sub WF-net, and system $S_s = ([N_s], [N_s, \bullet])$ is the potential sub WF-system.
3. Set $X$ is the set of labels only present in the potential sub WF-net, called the set of extending labels; that is, $X = \text{rng}(l_b) \setminus \text{rng}(l_b)$.
4. Sets $H$ and $B$ partition set $X$, where set $H$ is the set of extending labels that are hidden and set $B$ is the set of extending labels that are blocked.
5. Net $\Delta N_s$ is the potential sub WF-net after all labels in set $B$ have been blocked and all labels in set $H$ have been hidden, that is, $\Delta N_s = \tau_{H^0} \partial_B(N_s)$, and system $\Delta S_s$ is the system $(\Delta N_s, [\bullet N_s], [N_s, \bullet])$. Note that this system’s initial state and successful terminal state are inherited from the potential sub WF-system $S_s$.

Using these notations, we define the concept of constraints. A constraint is simply a pair of sets of extending labels. To allow a successful branching-bisimilarity check either some extending labels must be hidden from the first set (the hide set), or some extending labels must be blocked from the second set (the block set). This notion of constraints provides a very general and flexible way of expressing constraints on partitionings of extending labels.

**Definition 5.1. (Hide, block) Constraint**

Let $h, b \subseteq X$. The pair $(h, b)$ is a constraint if and only if for any $H$ and $B$: $(S_b \sim_h \Delta S_s) \Rightarrow \left(h \subseteq B \lor b \subseteq H\right)$. If $h = \emptyset$, then set $b$ is called a block constraint; if $b = \emptyset$, then set $h$ is called a hide constraint.

In other words, there is no partitioning of the set of extending labels $X$ into sets $H$ and $B$ such that

\[
\begin{align*}
\Delta S_s &\sim_{h_{\Delta S_s}} S_s \\
\Delta S_s &\sim_{b_{\Delta S_s}} S_s \\
\end{align*}
\]
Deciding life-cycle inheritance

1. the systems \( S_b \) and \( \Delta S_e \) are branching bisimilar,
2. all of the elements in the hide set \( h \) are blocked, and
3. all of the elements in the block set \( b \) are hidden.

Thus, if all elements of the hide set \( h \) are blocked, and if all elements of the block set \( b \) are hidden, then both systems cannot be branching bisimilar. Note that, by definition, both systems cannot be branching bisimilar if the pair \((\emptyset, \emptyset)\) is a constraint.

In the next two sections, we
1. derive constraints from the fact that in a base WF-system the successful terminal state is always reachable, whereas due to blocking extending labels in the potential sub WF-system, in this latter system, the successful terminal state might not always be reachable,
2. derive block constraints from the fact that cycles in the (short-circuited) potential sub WF-system should match a cycle in the (short-circuited) base WF-system, and
3. derive hide constraints from the fact that cycles in the (short-circuited) base WF-system should be matched by cycles in the (short-circuited) potential sub WF-system.

In both sections, the WF-nets as shown in Figure 5.2 are used to explain the issues in detail.

**FIGURE 5.2.** A base WF-net and a potential sub WF-net.
5.3 Successful termination

The labels that the exhaustive search algorithm operates on, that is, the set of extending labels, are, by definition, only present in the potential sub WF-net. Therefore, the reachability graph of the base WF-system will not be affected by blocking or hiding these labels. Because the base WF-net is a sound WF-net, the successful terminal state is reachable from any reachable marking in the base WF-system. Branching bisimilarity distinguishes successful termination from unsuccessful termination (deadlock), because of the third requirement on a branching bisimulation (see Definition 3.41 on page 68). As a result, the encapsulated and abstracted potential sub WF-system can only be branching bisimilar to the base WF-system if its successful terminal state is reachable from any of its reachable markings. Thus, we cannot allow the encapsulated and abstracted potential sub WF-system to deadlock.

Consider the base WF-net and the potential sub WF-net as shown in Figure 5.2 and recall that successful termination for both corresponding systems coincides with the markings \([p^{3}]\) and \([p^{9}]\). Because of what we mentioned above, we cannot allow a token to get stuck in places \(p^{4}, p^{5}, p^{6}, p^{7},\) or \(p^{8}\) of the encapsulated and abstracted potential sub WF-net. If we block labels \(D\) and \(E\), a token in place \(p^{5}\) will be stuck; if we block label \(F\), a token in place \(p^{6}\) will be stuck; if we block label \(G\), a token in place \(p^{7}\) will be stuck. Note that labels \(A\) and \(B\) are present in the base WF-net, that is, they cannot be blocked and a token cannot get stuck in places \(p^{4}\) or \(p^{8}\). In general, if the potential sub WF-net contains a place \(p \in P \setminus \{N, \bullet\}\) from which every outgoing arc is labeled with an extending label, then we should not block all labels on these outgoing arcs. Thus, such a set of extending labels indicates a possible hide constraint. In the example of Figure 5.2, the label sets \(\{D, E\}\), \(\{F\}\), and \(\{G\}\) seem to be possible hide constraints.

However, the attentive reader has noticed that we made some provisions. After encapsulation and abstraction of the potential sub WF-system of Figure 5.2, places \(p^{6}\) and \(p^{7}\) might not be markable. If we block label \(C\), place \(p^{6}\) will be unmarkable; if we block label \(E\) and either label \(C\) or label \(F\), place \(p^{7}\) will be unmarkable. If a place is unmarkable, we are allowed to block all outgoing labels. Thus, set \(\{F\}\) is not a hide constraint because of the partitionings where label \(C\) is blocked and set \(\{G\}\) is not a hide constraint because of the partitionings where either (i) labels \(C\) and \(E\) or (ii) labels \(E\) and \(F\) are blocked. Although sets \(\{F\}\) and \(\{G\}\) are not hide constraints, the pairs \((\{F\}, \{C\})\) and \((\{G\}, \{C, E, F\})\) are both constraints: If we block all labels of the first set, we have to block at least one of the labels of the second set.

In the remainder of this section, we formalize this idea of deriving constraints from the successful termination of sound WF-nets. All places
in the potential sub WF-net of which all output transitions are labeled with extending labels induce a constraint, where its hide set is the set of extending labels of all output transitions and its block set is a set of extending labels whose encapsulation may render the place unmarkable. The first main result shows that if we block all output transitions of some place and if it is still possible to prove a life-cycle-inheritance relationship, then this place has to be unmarkable.

Theorem 5.1. Let label sets $H$ and $B$ partition label set $X$ such that place $p \in P_s \setminus \{N_s\}$ is a place with for all $t \in (N_s, p) \cdot : l_s(t) \in B$. Then $S_b \sim_b \Delta S_s$ implies that place $p$ is unmarkable.

Proof. This proof follows the opposite direction: if we assume place $p$ is markable, then both systems cannot be branching bisimilar. For this, we use the fact that the successful terminal state in one of the systems is always reachable, whereas in the other system it is not.

1. Place $p$ is markable (assumption).
2. There exists a marking $M \in [\Delta S_s]$ such that $M(p) \geq 1$ (Result 1).
3. For all $t \in (N_s, p) \cdot : l_s(t) \in B$ (presupposition).
4. For all $M_1 \in [(\Delta N_s, M, [N_s \bullet])] : M_1(p) \geq 1$ (Result 2 and Result 3).
5. $[N_s \bullet] \not\in [(\Delta N_s, M, [N_s \bullet])]$ (Result 4 and $p \neq N_s$).
6. The system $S_b$ is sound (presupposition).
7. For all $M_2 \in [S_b] : [N_b \bullet] \not\in [(N_b, M_2, [N_b \bullet])]$ (Result 5, Result 7, and Definition 3.39 (Soundness)).
8. There exists no branching bisimulation $R$ which relates marking $M$ to any reachable marking in $[S_b]$ (Result 5, Result 7, and Definition 3.41 (Branching bisimulation)).
9. There exists no branching bisimulation $R$ which relates marking $[\bullet N_s]$ to marking $[\bullet N_b]$ (Result 8, both markings are the initial states of their systems).
10. Systems $S_b$ and $\Delta S_s$ are not branching bisimilar (Result 9 and Definition 3.42 (Branching bisimilarity)).

The properties of a sound WF-net (Requirement 3 of Definition 3.9 on page 47 and requirement 3 of Definition 3.39 on page 66) guarantee that all places are markable from the initial marking. Thus, only blocking transitions in a sound WF-net can make places unmarkable. We use a structural property to find the extending labels that, when blocked in the potential sub WF-system, might render a place with an induced constraint unmarkable. When one of these labels gets blocked, the associated constraint is satisfied. However, it is possible that after blocking some of these extending labels the place is markable after all.
If so, this can only result in a suboptimally pruned search space and is, thus, safe. It is for our purposes more important that the structural property is efficiently computable.

The basis for the block set of our constraints is the following: If the base WF-system and the encapsulated and abstracted potential sub WF-system are branching bisimilar, and if a place in the latter WF-net is unmarkable, then there must be a transition with

1. a blocked label,
2. a directed path of arcs leading to the unmarkable place,
3. all its input places markable (otherwise this transition would be dead and, thus, blocking its label would not make a difference), and
4. all its input places having at least one other output transition (to remove tokens possibly put into them).

Thus, a place inducing a constraint may become unmarkable if a transition satisfying the above four conditions exists. The block set is the set of labels of all transitions satisfying these conditions.

In the potential sub WF-net of Figure 5.2, the pair \(\{F\}, \{C\}\) is a constraint, the pair \(\{G\}, \{C, E\}\) is a constraint, and the pair \(\{D, E\}, \emptyset\) is a constraint (thus, \(\{D, E\}\) is a hide constraint). Note that, because of condition 4 mentioned above, the constraint \(\{\emptyset, \{\emptyset\}\}\) can be strengthened to \(\{G\}, \{C, E\}\) : if label F is blocked, it would only shift the problem (a token that gets stuck) from place p7 to place p6. Also note that label G needs to be hidden as long as either label C or label E is hidden, whereas the constraint \(\{\emptyset, \{\emptyset\}\}\) is satisfied as soon as either label C or label E is blocked. Although this is safe, it could lead to suboptimal pruning of the search tree.

**Theorem 5.2.**

Let label sets \(H\) and \(B\) partition label set \(X\) such that \(S_b \sim_{b} \Delta S_s\) and let place \(p\) be an unmarkable place. Then there exists a transition \(t \in T_s\) such that \(l_s(t) \in B\), \(tF_s^+p\), and for all places \(q \in \bullet(N_s, t)\) : (ii) place \(q\) is markable and (iii) \(|(N_s, q)\bullet| > 1\).

**Proof.**

This proof uses the above mentioned observations.

1. Net \(N_s\) is sound (presupposition).
2. Place \(p\) in system \(S_s\) is markable (Result 1, Definition 3.9 (WF-net) and Definition 3.39 (Soundness)).
3. Place \(p\) in system \(\Delta S_s\) is unmarkable (presupposition).
4. There exists a transition \(t \in T_s\) such that \(l_s(t) \in B\), \(tF_s^+p\), and for all \(q \in \bullet(N_s, t)\) : \(q\) is markable (Result 2, Result 3, Definition 3.43 (Encapsulation), and Definition 3.44 (Abstraction): Only blocking
a transition can make a place unmarkable and only blocking an enabled transition can make a difference).

5. Systems $S_b$ and $\Delta S_s$ are branching bisimilar (presupposition).

6. A token in a place $q$ as mentioned in Result 4 needs to be removable in $\Delta S_s$ (Result 4 and Result 5).

7. Place $q$ has more than one output transition (Result 4, Result 6: Transition $t$ is blocked and cannot remove a token from place $q$).

Theorem 5.1 implies that, if some place of which all output transitions have extending labels is markable, then from the specified set of labels $h$ at least one label must be hidden for a successful branching-bisimilarity check. Theorem 5.2 states that from a specified set of extending labels $b$ at least one label must be blocked if such a place is unmarkable. Thus, the pair $(h, b)$ is a constraint. However, one of the conditions in Theorem 5.2 is non-structural, non-efficiently computable, namely the one concerning the markability of the input places of the mentioned transition $t$ (condition 3). As explained before, weakening a constraint is safe, thus, we can replace this condition with the obvious requirement that transition $t$ is not an output transition of the unmarkable place $p$. Therefore, we obtain the following theorem.

**Theorem 5.3.** Suppose place $p \in P \setminus \{N_s, \star\}$ is a place such that for all transitions $t \in (N_s, p) \star : l_s(t) \in X$. Then, the pair $(h, b)$ is a constraint, where

1. $h = \{l_s(t) \mid t \in (N_s, p) \star \}$ and
2. $b = \{l_s(t) \mid t \in T_s \setminus (N_s, p) \star \land l_s(t) \in X \land t F_s \star p \land \forall q \in (N_s, t) \colon \#(N_s, q) \star > 1\}$

**Proof.**

This follows directly from Theorem 5.1 and Theorem 5.2.

When we take the Revised WF-net (see Figure 3.4 on page 46) as the base WF-net and the Original WF-net (see Figure 3.1 on page 38) as potential sub WF-net, the latter net contains three places that have only transitions with extending labels in its output set: $p_3$, $p_5$, and $p_6$. Place $p_3$ induces the hide set $\{\text{archive, reconnect}\}$. Because this place may be unmarkable if label confirm is blocked, it also induces the block set $\{\text{confirm}\}$. To explain this in-depth, Figure 5.3 shows the essence why blocking label confirm may render place $p_3$ unmarkable:

1. It shows that transition $t_7$ is labeled confirm.
2. It shows that there exists a path $(t_7, p_6, t_6, p_4, t_4, t_3)$ from transition $t_7$ to place $p_3$.
3. It shows that transition $t_7$ is not an output transition for place $p_3$.
4. It shows that both input places of transition $t_7$ have more than one output transition: place $p_6$ also has transition $t_6$ as output, place $p_5$ also has transition $t_3$ as output.

Thus, combining the hide set and the block set, place $p_3$ induces the constraint $\{\text{archive, reconnect}\}$, $\{\text{confirm}\}$. Likewise, place $p_5$ induces the constraint $\{\text{archive, confirm}\}$, $\{\text{reconnect}\}$.

5.4 Matching cycles

Recall that life-cycle inheritance is based on branching bisimilarity after hiding and blocking extending labels. A fundamental property of branching bisimilarity is that if some system can generate a certain sequence of observable labels, any other branching bisimilar system can generate the same sequence of observable labels. Cycles can generate infinite sequences of observable labels. Because sound WF-nets correspond to bounded systems, cycles are the only way to generate infinite sequences of observable labels. If some system contains a cycle, then any other branching bisimilar system should contain a cycle that can generate the same sequence of observable labels. As a result, the observable-label supports of these two cycles, that is, the sets of observable labels in the cycles, should be identical. Note that the cycles themselves need not be identical: Figure 5.4 shows a net that is branching bisimilar to the Original net shown in Figure 3.4 on page 46, but that does not contain a cycle (order). Instead, it contains a cycle (order, order), which is different from (order). (We obtained the net shown in Figure 5.4 by unfolding the cycle (order) once.) Nevertheless, if two systems have different observable-label supports for their cycles, they cannot be branching bisimilar, as Theorem 5.4 shows.
However, before presenting this theorem, we first need to define cycles and their observable-label supports.

**Definition 5.2.** Cycle

Let $S = (N, I, O)$ be a system and let $s$ be a firing sequence of net $N$. Sequence $s$ is a cycle of system $S$ if and only if there exists a marking $M \in S$ such that $M[s] \rightarrow M$. We use $\sigma S$ to denote the set of all cycles of system $S$.

**Definition 5.3.** Observable-label support of cycles

Let $N = (P, T, F, l)$ be a net, let $S = (N, I, O)$ be a system, and let $c = (t_1, t_2, t_3, \ldots)$ be a cycle of system $S$, that is, $c \in \sigma S$. The observable-label support of cycle $c$ is the set of observable labels occurring in cycle $c$, that is, $\{l(t_1), l(t_2), l(t_3), \ldots\}\{\tau\}$. We use $\langle c \rangle$ to denote the observable-label support of a cycle $c$. If set $C$ is a set of cycles, then we use $\langle C \rangle$ to denote the set of observable-label supports of all cycles in set $C$, that is, $\langle C \rangle = \{\langle c \rangle | c \in C\}$. Thus, $\langle \sigma S \rangle$ denotes the set of observable-label supports of all cycles of system $S$.

The next theorem states that two systems can only be branching bisimilar if the observable-label supports of their cycles match.

**Theorem 5.4.**

Let systems $S_1$ and $S_2$ be branching bisimilar. Then $\langle \sigma S_1 \rangle = \langle \sigma S_2 \rangle$.

**Proof.**

This proof follows the opposite direction: If one of the systems supports a cycle that the other does not support, then first system can exhibit behavior that the second system cannot mimic; thus, both systems cannot be branching bisimilar.

1. Assume $\langle \sigma S_1 \rangle \neq \langle \sigma S_2 \rangle$ (proof by contradiction).
Matching cycles

2. Assume, without loss of generality, that \( \langle \sigma S_1 \rangle \setminus \langle \sigma S_2 \rangle \) is not empty (Result 1).

3. Take a support \( v \in \langle \sigma S_1 \rangle \) such that \( v \not\in \langle \sigma S_2 \rangle \) (Result 2).

4. System \( S_1 \) can generate an infinite firing sequence containing exactly the labels from support \( v \), whereas system \( S_2 \) cannot (Result 3).

5. Systems \( S_1 \) and \( S_2 \) are not branching bisimilar (Result 4, Definition 3.41 (Branching bisimulation), and Definition 3.42 (Branching bisimilarity)).

As mentioned above, for two bounded systems to be branching bisimilar, the observable-label supports of their cycles should be equivalent. However, some parts of systems need not be covered by cycles, which reduces the effectiveness of this check. Fortunately, we are only interested in systems that correspond to sound WF-nets, and in corresponding short-circuited systems all parts are, by definition, covered by cycles. Therefore, it makes sense to compare the observable-label supports of the short-circuited WF-systems instead of the supports of the WF-systems themselves.

The next theorem states that two WF-systems can only be branching bisimilar if their short-circuited systems are branching bisimilar, and vice versa.

**Theorem 5.5.**

\[
(S_b \sim_b \Delta S_s) \iff (\varphi S_b \sim_b \Delta \varphi S_s), \text{ where } \Delta \varphi S_s = \tau_H \circ \partial_B (\varphi S_s).
\]

**Proof.**

This follows directly from Definition 3.11 (Short-circuited WF-net), Definition 3.41 (Branching bisimulation), and Definition 3.42 (Branching bisimilarity).

**Corollary 5.6.**

A potential sub WF-net can only be a subclass under life-cycle inheritance of a base WF-net if, after short-circuiting the systems and hiding and blocking all extending labels, the observable-label supports of their cycles are equivalent.

We can derive both hide and block constraints from this corollary, as the following subsections show.

**5.4.1 Block constraints**

Suppose we would hide all extending labels in the potential sub WF-net of Figure 5.2. After short-circuiting the resulting system, it has cycles with observable-label supports \( \emptyset \), \( \{A\} \), and \( \{B\} \), which the short-circuited base WF-net does not have. Clearly, if we want a suc-
Deciding life-cycle inheritance

successful branching-bisimilarity check, we should disable all cycles with observable-label supports \( \emptyset \), \( \{A\} \), or \( \{B\} \). The only way we can disable a cycle is by blocking at least one of its extending labels. If no extending labels are present in such a cycle, then we cannot disable it, in which case we have a constraint that cannot be satisfied: There is no partitioning of extending labels yielding branching bisimilarity. Otherwise, we can possibly achieve branching bisimilarity by blocking at least one of the extending labels present in every cycle with observable-label supports \( \emptyset \), \( \{A\} \), or \( \{B\} \). To prevent cycles with observable-label support \( \emptyset \), we should block at least one of the labels C, F, G, and H; to prevent cycles with observable-label support \( \{A\} \), we should block at least one of the labels D and H, and at least one of the labels E, G, and H; to prevent cycles with observable-label support \( \{B\} \), we should block at least one of the labels C, F, and G. As a result, we conclude that the label sets \( \{C, F, G, H\} \), \( \{D, H\} \), \( \{E, G, H\} \), and \( \{C, F, G\} \) are block constraints. The following is a direct corollary of Corollary 5.6. It states that if the set of extending labels in the observable-label support of any cycle in the short-circuited potential sub WF-system, is a block constraint if its set of base labels is not an observable-label support in the short-circuited base WF-system.

**Corollary 5.7.** Let \( \nu \in \langle \sigma \varphi S \rangle \) be a visible label support such that \( \forall X \notin \langle \sigma \varphi S \rangle \). The set \( \nu \cap X \) is a block constraint.

### 5.4.2 Hide constraints

Now, suppose we would block all extending labels in the potential sub WF-net of Figure 5.2. After short-circuiting the resulting system, it has no cycles, while the short-circuited base WF-system has a cycle with observable-label support \( \{A, B\} \). Clearly, if we want both systems to be branching bisimilar, we should hide some of the extending labels in such a way that cycles appear in the potential sub WF-net with the observable-label support \( \{A, B\} \). This can be done if such cycles exist when we would hide all extending labels. If no such cycle would exist, then both systems cannot become branching bisimilar for any partitioning of the extending labels. If such cycles would exist, we should not disable them all. As a result, we should hide all extending labels of one or more of these cycles. Considering the example of Figure 5.2, we should either (i) hide label D or (ii) hide labels E and G. This implies that the label set \( \{D, E, G\} \) is a hide constraint. Note that we do not use the fact that we need to hide both label E and label G if we would block label D. Because this information does not fit our notion of a hide or block constraint, we cannot use it at this point. However, as mentioned above, we prefer simplicity, wherever possible (as long as it leads to good results). The following is a second direct corollary of Corollary 5.6.
Matching cycles

Corollary 5.8. Let $v_b \in \langle \sigma \phi S_b \rangle$ be a visible label support such that $v_b \notin \langle \sigma \phi S \rangle$. The set $\{ l \in \nu \cap X | v_l \in \langle \sigma \phi S_b \rangle \land v_l \notin X = v_b \}$ is a hide constraint.

5.4.3 Structural properties

At this point, we deduced from the observable-label supports of all cycles in the two short-circuited WF-nets under consideration, a set of hide constraints and a set of block constraints. However, as explained above, we prefer to use structural properties rather than behavioral properties. Therefore, introduce the concepts of elementary cycles and elementary transition invariants and we argue that we can use elementary transition invariants instead of cycles. First, we argue that we only need to take minimal observable-label supports into account. Second, we argue that when taking only minimal observable-label supports into account, it suffices to take only elementary cycles into account. Third, we argue that, with some exceptions, we can replace elementary cycles by elementary transition invariants. We also discuss what to do when a system at hand happens to be one of these exceptions.

Minimal observable-label supports. A bag from a set of bags is called minimal if none of the other elements in the set is smaller than that bag.

Definition 5.4. Minimality

\[ s \subseteq \mu A \text{ is a set of bags over some alphabet } A. \text{ A bag } b \in s \text{ is minimal if and only if there exists no } b_1 \in s \text{ such that } b_1 < b. \text{ We use } \downarrow s \text{ to denote the set of all minimal bags of a set of bags } s. \]

The next theorem states that two systems can only be branching bisimilar if the minimal observable-label supports of their cycles should match.

Theorem 5.9. Let $S_1$ and $S_2$ be two bounded branching bisimilar systems. Then $\downarrow \langle \sigma S_1 \rangle = \downarrow \langle \sigma S_2 \rangle$.

Proof. This follows immediately from Theorem 5.4 and the fact that two sets can only be identical if their sets of minimal elements are identical.

Corollary 5.10. Let $v \in \downarrow \langle \sigma \phi S \rangle$ be a minimal observable-label support such that $v \cap X \notin \langle \sigma \phi S \rangle \setminus v \setminus X = v_b$. The set $v \cap X$ is a block constraint.
Corollary 5.11. Let \( v_b \in \left( \langle \sigma \psi S_b \rangle \right) \) be a minimal observable-label support such that \( v_b \not\in \left( \langle \sigma \psi S_b \rangle \right) \). The set \( \{ l \in v_b \cap X \} \) is a hide constraint.

Elementary cycles. The next definition relates firing sequences to bags of transitions.

Definition 5.5. Transition bag

Let \( S = (N, I, O) \) be a system, and let \( s = (t_1, t_2, t_3, \ldots) \) be a firing sequence of system \( S \). The transition bag \( b \in \mathbb{M}T \) of sequence \( s \) is the bag containing all transitions in sequence \( s \), that is \( b = [t_1] + [t_2] + [t_3] + \ldots \). We use \( \mathbb{M}s \) to denote the transition bag of sequence \( s \). If \( S \) is a set of sequences, we use \( \mathbb{M}S \) to denote the set of transition bags of all sequences in set \( S \).

Thus, the firing sequence \((t_1, t_5, t_7, t_6, t_5, t_6, t_8, t_4, t_3)\) (which is a firing sequence of the short-circuited Revised net) has the bag \([t_1, t_3, t_4, t_5, t_6, t_7, t_8, t_9, t_10, t_11, t_12, t_13, t_14] \) as its transition bag.

Definition 5.6. Elementary cycle

Let \( S = (N, I, O) \) be system and let \( c = (t_1, t_2, t_3, \ldots) \) be a cycle of system \( S \), that is, \( c \in \sigma S \). Cycle \( c \) is elementary if and only if its transition bag \( \mathbb{M}c \) is minimal in the set of transition bags of all cycles of \( \sigma S \).

We use \( \{ \sigma S \} \) to denote the set of all elementary cycles in system \( S \).

It is straightforward to check that every cycle is a combination of elementary cycles. The next theorem states that for every minimal observable-label support there exists an elementary cycle with that support.

Theorem 5.12. Let \( S \) be a system and let \( \nu \in \left( \langle \sigma S \rangle \right) \). There exists an elementary cycle \( c \in \left( \langle \sigma S \rangle \right) \) such that \( \langle c \rangle = \nu \).

Proof. This proof uses the fact that there has to exist a cycle supporting \( \nu \), that this cycle is a combination of elementary cycles, and that these elementary cycles have to have support \( \nu \) (because \( \nu \) is minimal).

1. The set \( \nu \in \left( \langle \sigma S \rangle \right) \) (presupposition).
2. There exists a cycle \( c_1 \in \sigma S \) such that its support \( \langle c_1 \rangle = \nu \) (Result 1 and Definition 5.3 (Observable-label support of cycles)).
3. There exists a \( c \in \left( \sigma S \right) \) such that the transitions in cycle \( c \) form a strict subbag of the transitions in cycle \( c_1 \) (Result 2 and Definition 5.6 (Elementary cycle)).
4. The set \( \langle c \rangle \subseteq \langle c_1 \rangle \) (Result 3 and Definition 5.3 (Observable-label support of cycles)).
5. The set \( \langle c \rangle = \nu \) (Result 1, Result 2, and Result 4: the set \( \langle c \rangle \) is minimal).
Matching cycles

As a result of Theorem 5.12, we only need to take into account elementary cycles when taking into account only minimal observable-label supports. Thus, we can replace Corollary 5.10 and Corollary 5.11 by Corollary 5.13 and Corollary 5.14.

Corollary 5.13.

Let \( v \in \left\{ \left[ \sigma \phi S_a \right] \right\} \) be a visible label support of an elementary cycle such that \( v \setminus X \in \left\{ \left[ \sigma \phi S_b \right] \right\} \). The set \( v \cap X \) is a block constraint.

Corollary 5.14.

Let \( v_b \in \left\{ \left[ \sigma \phi S_b \right] \right\} \) be a visible label support of an elementary cycle such that \( v_b \notin \left\{ \left[ \sigma \phi S_a \right] \right\} \). The set \( \{ l \in v_s \cap X | v_s \in \left\{ \left[ \sigma \phi S_s \right] \right\} \wedge v_s \setminus X = v_b \} \) is a hide constraint.

Elementary transition invariants. We use \( \sigma N \) to denote the set of transition invariants of a net \( N \).

Definition 5.7.

Elementary transition invariant

Let \( N = ( P, T, F, l ) \) be a net and let \( w \in \mu T \) be a transition invariant of net \( N \), that is, \( w \in \sigma N \). Transition invariant \( w \) is elementary if and only if it is minimal in \( \sigma N \), that is, if and only if there exists no \( w_1 \in \sigma N \) such that \( w_1 < w \). We use \( \left\{ \sigma N \right\} \) to denote the set of all elementary transition invariants in net \( N \).

The set of transition invariants of the short-circuited Revised net (see Figure 3.5 on page 48) can be described by the set \( \{ t_1^a + t_2^b + t_3^a + t_4^a + t_5^c + t_6^c + t_7^d + t_8^d + t^e | a, b, c, d, e \in \mathbb{N} \} \).

From this, we conclude that the transition invariants \( t_1 + t_3 + t_4 + t \), \( t_2 + t_4 \), \( t_5 + t_6 \), and \( t_7 + t_8 \) are elementary.

The fundamental difference between transition invariants and cycles, is that the latter takes the fireability of the constituent transitions into account, whereas the former does not. Figure 5.5 shows a WF-net visualizing this difference. It is straightforward to check that the short-circuited net contains the transition invariants \( t_1 + t_6 + t_8 + t_9 + t \) and \( t_3 + t_4 + t_6 + t_11 + t \), where \( t \) is the short-circuited transition. However, the corresponding short-circuited system does not contain cycles that have matching transition bags. For example, the firing sequence \( (t_1, t_6, t_8, t_9, t) \) is not a cycle, because after having fired transition \( t_1 \), transition \( t_6 \) will not be enabled because place \( p_3 \) is unmarked.

Definition 5.8.

Observable-label support of transition invariants

Let \( N = ( P, T, F, l ) \) be a net and let \( w \in \sigma N \). The observable-label support of \( w \) is the set of observable labels occurring in \( w \): \( \{ I(t) | t \in T \wedge w(t) > 0 \} \setminus \{ \tau \} \).

We use \( \langle w \rangle \) to denote the observable-label support of transition invariant \( w \). If \( W \) is a set of transition invariants, we use \( \langle W \rangle \) to denote the set of observable-label supports of all transition invariants in \( W \).
Deciding life-cycle inheritance

It is easy to check that, by definition, every cycle induces a transition invariant. For sound free-choice nets, the reverse is also true [DE95]. Unfortunately, as explained above, for sound non-free-choice nets, this need not be true: Figure 5.5 shows a sound WF-net that, when short-circuited, contains transition invariants that do not induce cycles. As a result, the set of transition bags of cycles might be a strict subset of the set of transition invariants: $\mu_\sigma S < \sigma N$, where system $S$ is an arbitrary system of net $N$. However, in the context of WF-nets, we expect such a situation (a transition invariant that does not induce a cycle) to be extremely rare. Therefore, we assume that every transition invariant does induce a cycle. Under this assumption, $\mu_\sigma S = \sigma N$, and, thus, $\mu_\sigma S = [\sigma N]$, and we can replace Corollary 5.13 and Corollary 5.14 by Corollary 5.15 and Corollary 5.16.

**Corollary 5.15.**

Assume $\mu_\sigma S = [\sigma \phi N_b]$ and $\mu_\sigma S = [\sigma \phi N_s]$. Let $\nu \in [\langle \sigma \phi N_b \rangle]$ be a minimal observable-label support of an elementary transition invariant such that $\forall X \in [\langle \sigma \phi N_b \rangle]$. The set $\nu \cap X$ is a block constraint.

Thus, if in both systems every transition invariant induces a cycle, then any transition invariant of the potential sub WF-net that is not matched
Matching cycles

by any transition invariant of the base WF-net, induces a block constraint: The cycle corresponding to this transition invariant cannot be matched by a cycle in the base WF-system and should, thus, be prevented from occurring.

**Corollary 5.16.** Assume \( \mu[\sigma \varphi S_b] = \{\sigma \varphi N_b\} \) and \( \mu[\sigma \varphi S_s] = \{\sigma \varphi N_s\} \). Let \( v_b \in \{\langle \{\sigma \varphi N_b\}\rangle\} \) be a minimal observable-label support of an elementary transition invariant such that \( v_b \notin \{\langle \{\sigma \varphi N_s\}\rangle\} \). The set \( \{t \in v_s \cap X \mid v_s \notin \{\langle \{\sigma \varphi N_s\}\rangle\} \land v_s \not\in v_b \} \) is a hide constraint.

Thus, if in both systems every transition invariant induces a cycle, then any transition invariant of the base WF-system that is not matched by any transition invariant in the potential sub WF-system, induces a hide constraint: Some cycle corresponding to some transition invariant in the potential sub WF-net that is larger than the transition invariant in the base WF-net, should be able to occur.

For computing elementary transition invariants, a reasonably efficient algorithm exists [CS90]. However, if the assumption is not correct, we might incorrectly prune possible solutions from the search tree. Therefore, if the backtracking algorithm does not find a solution, we need to check whether the assumption holds, that is, we need to check whether every elementary transition invariant induces an elementary cycle. First, we check whether the base WF-net and the potential sub WF-net are free choice. If so, the assumption holds. Otherwise, we need to check whether every elementary transition invariant induces an elementary cycle using the reachability graphs of both WF-nets. If the assumption holds, there is indeed no solution, otherwise, we need to update the constraints or simply give way to an exhaustive search algorithm for the partitionings that have not yet been checked to find a solution, if any.

**5.4.4 Example**

When we take the Original net as the base WF-net of Figure 3.4 on page 46 and the Revised net of Figure 3.1 on page 38 as potential sub WF-net, applying Corollary 5.15 and Corollary 5.16 yields the following results.

The short-circuited Original net contains the elementary transition invariants \( t_1 + t_2 + t, t_4 + t_5, t_6 \), and \( t_1 + t_4 + t_7 + t_3 + t \), which are all matched by elementary cycles. The observable-label supports of these invariants are \{connect, disconnect\}, \{order, ship\}, \{order\}, and \{connect, disconnect, order, ship\}. It is straightforward to check that second support contains the third and that the fourth contains the first. As a result, only the first and the third support are minimal.
Deciding life-cycle inheritance

The short-circuited Revised net contains the elementary transition invariants $t_1 + t_4 + t_3 + t$, $t_2 + t_4$, $t_5 + t_6$, and $t_7 + t_8$, which are all matched by elementary cycles. Their observable-label supports are \{archive, connect, disconnect\}, \{disconnect, reconnect\}, \{order\}, and \{confirm, ship\}. All these supports are minimal.

When projecting the supports of the short-circuited Implementation net to the base labels, we obtain the supports \{connect, disconnect\}, \{disconnect\}, \{order\}, and \{ship\}. It is clear, that the supports \{disconnect\} and \{ship\} cannot be matched by any of the supports of the short-circuited Original net. Therefore, we should block labels reconnect and confirm. Furthermore, to be able to match the support \{connect, disconnect\} of the Original net, we should hide label archive. Thus, we obtain the block constraints \{reconnect\} and \{confirm\}, and the hide constraint \{archive\}.

5.5 Backtracking algorithm

The backtracking algorithm prunes the search tree when possible. For this pruning, we use the constraints as described in the previous two sections. The basic algorithm is shown in Figure 5.6. First, we compute the constraints (line 1) (For computing the transition invariants we use the algorithm as described in Section 6 of [CS90]. Note that it suffices to generate elementary transition invariants.) Second, we initialize two types of counters:

1. For every constraint a counter named $\text{HideCtr}$, which counts the number of labels in the hide set that are either hidden or unassigned. Initially, this counter is set to the size of the hide set. (Whenever an element from this set is blocked, the counter is decremented.)

2. For every constraint a counter named $\text{BlockCtr}$, which counts the number of labels in the block set that are either blocked or unassigned. Initially, this counter is set to the size of the block set. (Whenever an element from this set is hidden, the counter is decremented.)

Using these counters, it is very efficient to check whether a constraint is violated. A constraint $c = (h, b)$ is violated if all elements of its hide set are blocked ($\text{HideCtr}[c] = 0$) and if all elements of its block set are hidden ($\text{BlockCtr}[c] = 0$). If some of the constraints are violated from the beginning (line 2), we return without a solution (line 3).

In case the constraints are satisfiable, we initialize some necessary variables (line 5). Set $h$ refers to the set of extending labels that are hid-
Backtracking algorithm

1 compute constraints
2 if (constraint (∅, ∅) exists) {
3   return without solution
4 }
5 set $h$ to $∅$, $b$ to $∅$, $backtrack$ to false, and $n$ to $|X|$  // No label is hidden or blocked.
6 while (true) {
7   if (some constraint is violated by $h$ and $b$) {
8     set $backtrack$ to true
9   } else {
10      if ($n$ equals 0) { // All labels are hidden or blocked.
11         set $H$ to $h$ and $B$ to $b$
12         if (branching bisimilar) {
13           return with solution
14         }
15         set $backtrack$ to true  // Backtrack to next assignment.
16       } else {
17         decrement $n$  // Block first unassigned label
18         add $X[n]$ to $b$  // (we prefer blocking over hiding).
19       }
20   } if ($backtrack$) {
21      if ($b$ is empty) { // No label is blocked, thus, we cannot
22        return without solution  // reassign any label.
23      }
24      while ($X[n]$ in $h$) { // Some labels are blocked.
25        remove $X[n]$ from $h$  // Unassign trailing hidden labels, hiding them
26        increment $n$  // has no use at the moment.
27      }
28      move $X[n]$ from $b$ to $h$  // Hide last blocked label.
29      set $backtrack$ to false
30   }

FIGURE 5.6. The basic backtracking algorithm.

den, set $b$ refers to the set of extending labels that are blocked, boolean $backtrack$ indicates whether we detected a dead end, and number $n$ equals the number of extending labels still to assign. When number $n$ equals 0, sets $h$ and $b$ partition set $X$ and are therefore valid candidates for sets $H$ and $B$. Note that a partial assignment of labels to sets $h$ and $b$ may already violate some constraint, namely if $h$ is fully contained in set $B$ and $b$ is fully contained in set $H$. It is clear that it is of no use to extend a partial assignment that violates some constraint: A violated constraint will remain violated.

Repeatedly, we check the current assignment, sets $h$ and $b$ (lines 6–30). If the current assignment violates some constraint, we decide to backtrack (line 8). If the current assignment did not violate some constraint, we check whether the assignment corresponds to a partitioning of set $X$
Deciding life-cycle inheritance

If so, we compute both reachability graphs (if not yet computed, see below), set sets \( H \) and \( B \) to sets \( h \) and \( b \), and check branching bisimilarity between systems \( S_s \) and \( \Delta S_s \) (line 12). If branching bisimilar, we return with the current partitioning as a solution (line 13); otherwise, we decide to backtrack (line 15). If the current assignment was only partial, we block the next unassigned extending label (lines 16–18), generating a new assignment for the next iteration. Note that the backtracking algorithm assumes some kind of total ordering on the labels in set \( X \). Also note that the backtracking algorithm prefers to block labels (because blocking reduces the effective size of the reachability graph where hiding has no effect on this size). Thus, the last partitioning to check is the one that corresponds to assigning all extending labels in set \( X \) to set \( h \). Furthermore, if an assignment assigning only labels to set \( h \) violates any constraints, no solution is possible anymore.

When it was decided to backtrack, we first check whether there are assignments left to check (line 21). If not, we return without a solution (line 22). Otherwise, we repeatedly unassign the last assigned extending label until we find a blocked label (lines 24–27), hide this blocked label (line 28), and reset the backtrack variable (line 29).

The reachability graphs computed are the reachability graphs of the base WF-system and the potential sub WF-system. Both graphs are computed only once and only when necessary. Arcs in the reachability graph of the potential sub WF-system corresponding to blocked labels are not traversable by the algorithm that checks branching bisimilarity, while arcs corresponding to hidden labels are traversable but not observable by this algorithm.

Figure 5.6 shows how the backtracking algorithm performs when we take the Specification net (see Figure 3.4 on page 46) as base WF-net and the Implementation net (see Figure 3.1 on page 38) as potential sub WF-net. Table 5.1 summarizes the constraints we found in the previous sections. The arrows in Figure 5.6 indicate the actual tree tra-

<table>
<thead>
<tr>
<th>Hide set</th>
<th>Block set</th>
</tr>
</thead>
<tbody>
<tr>
<td>{archive, reconnect}</td>
<td>{confirm}</td>
</tr>
<tr>
<td>{archive, confirm}</td>
<td>{reconnect}</td>
</tr>
<tr>
<td>{archive}</td>
<td>\Ø</td>
</tr>
<tr>
<td>\Ø</td>
<td>{reconnect}</td>
</tr>
<tr>
<td>\Ø</td>
<td>{confirm}</td>
</tr>
</tbody>
</table>

TABLE 5.1. All constraints for the Original net and the Revised net.
versal of the backtracking algorithm, using the hide and block constraints as displayed in Table 5.1. In contrast to the exhaustive search algorithm (see Figure 5.1), the backtracking algorithm needs only one branching bisimilarity check before concluding that no solution exists.

FIGURE 5.6. The backtracking algorithm in action.

5.6 Case study

To test the effectiveness and efficiency of the backtracking algorithm, we asked, without offering any explanations or limitations, an assistant professor in our department to take a number of WF-nets made by workflow students (base WF-nets) and to add new functionality to these WF-nets, yielding the same number of potential sub WF-nets. We took these pairs of WF-nets as samples to compare our backtracking algorithm to an exhaustive search algorithm (the backtracking algorithm with an empty set of constraints). Table 5.2 shows the test results. The top rows of the table show

- the number of places,
- the number of transitions, and
- the number of reachable states

for both WF-nets, the middle rows show

- the number of possible partitionings,
- the number of partitionings that satisfy all constraints,
- the number of partitionings that yield branching bisimilarity (solutions),
- the number of branching bisimilarity checks needed by the backtracking algorithm (BA), and
Deciding life-cycle inheritance

...the number of branching bisimilarity checks needed by the exhaustive search algorithm (ESA), and the bottom rows shows the measured computation times for both algorithms.

Recall that our main goal is to reduce the number of branching bisimilarity checks to prevent excessive processing times. For six out of seven samples, the number of branching bisimilarity checks needed by the backtracking algorithm to find a solution is reduced to an absolute minimum: one for samples A, C, D, E, and F, and none for sample B. For sample G, we need to check branching bisimilarity twice before concluding that there exists no solution. Note that although this is not optimal, it is far better than the number of branching bisimilarity checks needed by the exhaustive search algorithm (eight). We conclude that, for the samples, the backtracking algorithm is very effective.

For four out of seven samples, the exhaustive search algorithm outperforms the backtracking algorithm when considering computation time. In our opinion, this is acceptable, because none of the samples is very complex. In the test samples, the overhead for computing the constraints make a comparison of run-times not very meaningful. Recall that the problem we try to tackle with the backtracking algorithm is the

<table>
<thead>
<tr>
<th>Base OLC</th>
<th>A1</th>
<th>B1</th>
<th>C1</th>
<th>D1</th>
<th>E1</th>
<th>F1</th>
<th>G1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Places</td>
<td>36</td>
<td>22</td>
<td>19</td>
<td>18</td>
<td>20</td>
<td>19</td>
<td>30</td>
</tr>
<tr>
<td>Transitions</td>
<td>32</td>
<td>24</td>
<td>19</td>
<td>16</td>
<td>25</td>
<td>21</td>
<td>37</td>
</tr>
<tr>
<td>Reachable states</td>
<td>469</td>
<td>24</td>
<td>30</td>
<td>22</td>
<td>21</td>
<td>20</td>
<td>30</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Potential sub OLC</th>
<th>A2</th>
<th>B2</th>
<th>C2</th>
<th>D2</th>
<th>E2</th>
<th>F2</th>
<th>G2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Places</td>
<td>37</td>
<td>25</td>
<td>19</td>
<td>22</td>
<td>20</td>
<td>20</td>
<td>34</td>
</tr>
<tr>
<td>Transitions</td>
<td>35</td>
<td>30</td>
<td>22</td>
<td>18</td>
<td>29</td>
<td>24</td>
<td>42</td>
</tr>
<tr>
<td>Reachable states</td>
<td>477</td>
<td>29</td>
<td>30</td>
<td>30</td>
<td>21</td>
<td>23</td>
<td>34</td>
</tr>
</tbody>
</table>

### Partitionings
- satisfying constraints: 1 0 1 1 1 1 2 2
- Solutions: 1 0 1 1 1 2 0
- BB checks: BA 1 0 1 1 1 1 2
- ESA 3 16 1 4 1 1 8

<table>
<thead>
<tr>
<th>Time: BA (in s/100)</th>
<th>536.8</th>
<th>0.31</th>
<th>0.52</th>
<th>0.29</th>
<th>1.28</th>
<th>0.37</th>
<th>3.83</th>
</tr>
</thead>
<tbody>
<tr>
<td>— ESA (in s/100)</td>
<td>1654.2</td>
<td>2.19</td>
<td>0.36</td>
<td>0.81</td>
<td>0.12</td>
<td>0.13</td>
<td>2.82</td>
</tr>
</tbody>
</table>

TABLE 5.2. Results on the test samples.
Conclusions

combination of large state spaces (that is, up to millions of states for workflows) and the exponential factor in the number of possible partitionings. From all samples, sample A is, by far, the most typical. It is satisfactory to see that for this sample, we reduce the computation time from over sixteen seconds to less than six. Of course, we would like to test the backtracking algorithm on more, and more complex, samples before concluding that it is efficient in general.

5.7 Conclusions

We have introduced a backtracking algorithm that, when compared to an exhaustive search algorithm, seems to be able to prevent excessive processing times when checking life-cycle inheritance [Bas98, BA01, AB02]. By exploiting structural properties of WF-nets, this backtracking algorithm constructs so-called constraints to avoid checking branching bisimilarity for every possible combination of blocking and hiding extending labels.

We have made no attempts to improve either the algorithm that checks branching bisimilarity or the algorithm that generates the reachability graph, as in our eyes both algorithms are optimal. As a result, we also have made no attempts to improve the algorithm to check any of the other inheritance relations (protocol, projection, and protocol/projection), as both deciding protocol inheritance and deciding projection inheritance need only to generate both reachability graphs and check branching bisimilarity once, and deciding protocol/projection needs only to generate both reachability graphs and check branching bisimilarity twice. In contrast with this, life-cycle inheritance needs to generate both reachability graphs and may need checking branching bisimilarity a considerable number of times.

The backtracking algorithm only works for WF-nets, because, as mentioned above, it uses structural properties of WF-nets to avoid branching bisimilarity checks. However, if we can map these properties onto different formalisms, then we should be able to use the techniques the algorithm is based on for those different formalisms. For example, if we can detect deadlocks and/or elementary cycles in process algebras without generating a state space, then we might be able to use the backtracking algorithms for process-algebra-based process definitions too.

Extreme examples exist for which the basic backtracking algorithm fails in finding a solution, although a solution exists. This is due to the fact that we base our constraints only on structural properties. As a result, these constraints might not be supported by the corresponding state spaces. However, we believe that such an example does not occur
in practice. Nevertheless, if such an example is encountered, that is, after the backtracking algorithm has failed to find a solution and if the computed constraints are not supported by the corresponding reachability graphs, then we simply check whether any combination that violated the constraints is a solution (thus, in this case, the backtracking algorithm degenerates to an exhaustive search algorithm).

The backtracking algorithm has been implemented in a prototype of the next version of Woflan (Woflan 3). The current version of Woflan (Woflan 2.2) implements only an exhaustive search algorithm. The reason for implementing the algorithm in a prototype is that its performance heavily relies on an efficient algorithm to compute elementary transition invariants (because the backtracking algorithm uses these invariants to compute constraints). The existing versions of Woflan all use a generic equation solver to compute these invariants, although more efficient algorithm were known in literature. The prototype combines the backtracking algorithm with the efficient invariant algorithm from [CS90].
The previous three chapters introduce (WF-)nets and show that relevant properties (soundness, life-cycle inheritance) can be decided ‘in a reasonably’ efficient way. However, these chapters all assume that a WF-net is used to model a workflow process definition. As we do not expect this assumption to hold, we need to demonstrate that it is possible to successfully map the behavior of existing workflow process definitions onto WF-nets. This chapter demonstrates that this is possible in general, whereas the next chapter shows that this is possible for workflow process definitions from some specific workflow management systems. Furthermore, it shows that in some cases we can obtain additional soundness results.

To demonstrate that we can successfully map the behavior of existing workflow process definitions onto WF-nets in general, the eXchangeable Routing Language (XRL) [AK02] is used as a proof-of-concept. The reasons for selecting XRL are threefold:

1. XRL is based on the most relevant patterns found in the workflow management field today [AHKB03]. Thus, if we can map any XRL process onto a WF-net, then we can map the most relevant workflow patterns onto WF-nets.
2. XRL is instance-based, and changes can be made almost at any point in time to an XRL process. Thus, verification of an XRL process before enacting it is of the utmost importance.
3. XRL is extendible: an extension can be added to XRL by providing the semantics for it. Verification of an extension before using it in any XRL process is also of the utmost importance.
The remainder of this chapter is organized as follows. Section 6.1 gives a short motivation for the development of XRL. Section 6.2 presents the parts of XRL that are relevant for this thesis, that is, the part of XRL that specifies the control flow. Section 6.3 presents the WF-net mapping of an XRL process definition. As a result, we can apply the diagnosis process as introduced in Chapter 4 to XRL process definitions. Section 6.4 discusses interesting soundness results for XRL process definitions. For example, it shows that only certain XRL constructs can violate the soundness property, and that we can use this result and the reduction rules presented in Chapter 4 to reduce the resulting WF-net before verifying it. Section 6.5 concludes the chapter.

6.1 Motivation

Today’s corporations often must operate across organizational boundaries. Phenomena such as E-commerce, extended enterprises, and the Internet, stimulate cooperation between organizations. Therefore, the importance of workflows distributed over a number of organizations is increasing [Aal00a, Aal00b, GAHL01, LASS01]. Inter-organizational workflow offers companies the opportunity to re-shape business processes beyond the boundaries of their own organizations. However, inter-organizational workflows are typically subject to conflicting constraints. On the one hand, there is a strong need for coordination to optimize the flow of work in and between the different organizations. On the other hand, the organizations involved are essentially autonomous and have the freedom to create or modify workflows at any point in time. These conflicting constraints complicate the development of languages and tools for cross-organizational workflow support.

Looking at existing initiatives, it can be noted that until recently:

- process support for cross-organizational workflow has been neglected since the lion’s share of attention has gone to data and
- only mainly pre-specified standardized processes have been considered (such as, market places, procurement, and so on).

Recently, a number of standards have been proposed that all aim to provide this kind of cross-organizational support. Examples of these proposed standards include the Business Process Execution Language for Web Services (BPEL4WS), the Business Process Modeling Language (BPML), Web Service Choreography Interface (WSCI), Electronic Business using eXtensible Markup Language (ebXML), and XML Process Definition Language (XPDL) [Aal03a]. However, these standards are mainly driven by existing commercial products, and often lack an unambiguous semantics.
Motivation

Based on these observations, XRL has been developed. The idea to develop a language like XRL was raised in [KZ02] and the definition of the language was given in [AK02]. XRL uses the syntax of XML, but contains constructs that embed the semantics of control flow. Moreover, XRL supports highly dynamic one-of-a-kind workflow processes. For example, the “first trade problem” is considered, that is, the situation where parties have no prior trading relationship [Lee99]. Clearly, the “first-trade problem” is the extreme case of highly dynamic one-of-a-kind workflow processes and therefore also the most difficult. To support highly dynamic one-of-a-kind workflow processes, XRL describes processes at the instance level, whereas traditional workflow modeling languages describe processes at the class or type level [JB96, Law97]. As a result, an XRL process definition describes the partial ordering of tasks for one specific instance. The advantages of doing so are threefold:

- An instance can be exchanged more easily, because it includes its definition.
- An instance can be modified without causing any problems for other instances, because instances are completely self-contained.
- The expressive power is increased, because we can tune a definition towards its instance. For example, we can easily handle a variable number of parallel or alternative branches by simply adding these branches to the instance.

The other side of the picture is that we have additional overhead, and management information is harder to obtain. The fact that the schema can be exchanged more easily outweighs, in our opinion, the additional overhead. The management information disadvantage can be lessened for a great deal when we restrict ourselves to changes that preserve a number of inheritance relations (see, for example [AB02]).

In research on workflow patterns [AHKB03, Aal03a, Aal03b], the expressive power
- of many contemporary workflow management systems, including COSA, HP Changengine, Forté Conductor, I-Flow, InConcert, MQSeries Workflow, R/3 Workflow, Staffware, Verve, and Visual WorkFlo, and
- of most of the proposed standards, including BPEL4WS, BPML, WSCI, and XPDL,

has been compared using a set of workflow patterns. Based on the workflow patterns supported by these systems, and their relative use in practice, the most relevant constructs have been carefully selected for XRL. As a result, the expressive power of XRL far exceeds that of each of the workflow management systems mentioned above.
As was shown in [AK02], the semantics of XRL process definitions can be expressed in terms of (Petri) nets. Unfortunately, this semantics does not yield WF-nets and, therefore, does not allow for a direct use of WF-net-based theoretical results and tools. This limitation was recognized in [AVK01a]. In this chapter, we present a direct mapping from XRL process definitions onto WF-nets, that is, the semantics of XRL process definitions are given in terms of WF-nets. This mapping has been implemented in XSLT (eXtensible Stylesheet Language Transformations).

6.2 An XML-based routing language

The focus of this thesis is on the verification of WF-nets. Therefore, we limit ourselves to only a brief introduction to XRL. For more details, we refer to [AK02, AVK01a, AVK01b, VAK04].

6.2.1 Syntax of XRL

The syntax of XRL is completely specified by a Data Type Definition (DTD) [BPSMM00], which is given in Appendix D. Alternatively, an XML Schema Definition can be used; see Appendix D for a link to this Schema. Figure 6.1 shows the (for this thesis) relevant parts of the DTD as a UML class diagram. An XRL process definition is a consistent XML document, that is, a well-formed and valid XML file [BPSMM00], with top element route. This route element contains exactly one child routing element, and:

1. passes control to this routing element immediately after the instance has been started and
2. completes the instance immediately after this routing element has returned control.

A routing element is an important building block of XRL. It can either be simple or complex. A simple routing element cannot contain routing elements, and can either be a terminate or a task.

**Terminate.** The entire instance has to be completed as soon as possible, that is, all routing elements have to return control as soon as possible.

**Task.** The associated work item is offered to the given resources, after which the task element waits until some resource has activated the work item and completed the corresponding activity, whereupon the task element sets all associated events and returns control. The purpose of having a task element setting an event is to be able to check later on
An XML-based routing language

**FIGURE 6.1.** Relevant parts of the XRL DTD.
in the process, using a wait element, whether or not this task has been executed or not.

A complex routing element can contain routing elements, and can either have one or more child routing elements, or exactly one child routing element, or be a condition. (Note that the true and false elements shown in Figure 6.1 are not routing elements. Therefore, a condition element cannot contain child routing elements, although it can contain descending routing elements.) A complex routing element containing one or more child routing elements can either be a sequence, an any_sequence, a choice, a parallel_sync, a parallel_no_sync, a parallel_part_sync, or a parallel_part_sync_cancel.

**Sequence.** Control is first passed to the first child routing element. If the first child routing element has returned control, then control is passed to the second child routing element, and so on. If the last child routing element has returned control, the sequence returns control. Thus, a sequence takes care that all its child routing elements are be performed, and that they are performed in the order specified.

**Any_sequence.** Control is passed to all child routing elements. However, the any_sequence element takes care that the execution of the child routing elements take place under mutual exclusion. If all child routing elements have returned control, the any_sequence returns control. Thus, an any_sequence takes care that all its child routing elements are be performed, and that they are performed in some order.

**Choice.** Control is passed to the first available child routing element. If multiple child routing elements are available, then the choice elements picks selects one. If this child routing element has returned control, the choice returns control. Thus, a choice takes care that only the first available child routing element is performed. For example, if all child routing elements are tasks, then the choice takes care that only the first task (work item) that gets activated gets performed (all other work items are withdrawn by the choice in such a case).

**Parallel_sync.** Control is passed to all child routing elements. If all child routing elements have returned control, the parallel_sync returns control. Thus, a parallel_sync takes care that all its child routing elements are be performed, and that they can be performed in parallel.

**Parallel_no_sync.** Control is passed to all child routing elements, and control is returned immediately. Thus, a parallel_no_sync takes care of spawning all its child routing elements in parallel. Note it does not synchronize these spawned routing elements.
Parallel_part_sync. Control is passed to all child routing elements. If the given number of child routing elements have returned control, the parallel_sync returns control. Thus, a parallel_part_sync element takes care that all its child routing elements are performed and can be performed in parallel, but that only a number of these child routing elements need to have returned control to return control. Note that a parallel_part_sync generalizes both the parallel_sync element and the parallel_no_sync element: if the given number equals the number of child routing elements, then the parallel_part_sync behaves identical to a parallel_sync, and if the given number equals 0, then the parallel_part_sync behaves identical to a parallel_no_sync.

Parallel_part_sync_cancel. A parallel_part_sync_cancel element is identical to a parallel_part_sync element, except for the fact that child routing elements that have not started yet are withdrawn (or cancelled) when control is returned. As a result, the parallel_part_sync_cancel element generalizes the parallel_sync element, but not the parallel_no_sync element: If the given number equals 0, the parallel_part_sync_cancel element might withdraw all its child routing elements.

Wait_all. A wait_all element waits until either:

- all associated events are set, that is, wait until all corresponding tasks have been executed,
- a child timeout element has taken over control, or
- a terminate element has occurred.

A child timeout element can take over control if, for example, its deadline expires. If a child timeout element takes over control, and if the child timeout element contains a child routing element, then the child timeout element passes control to this child routing element. If this child routing element has returned control, the child timeout element returns control to the wait_all element. If a child timeout element has taken over control, and if this child timeout element does not contain a child routing element, then the child timeout element immediately returns control to the wait_all element. If all associated events have been set, if a child timeout element has returned control, or if a terminate element has occurred, then the wait_all returns control.

Wait_any. A wait_any element is identical to a wait_all element, except for the fact that a wait_any element waits until any associated event has been set (instead of all). Thus, after a wait_any element has returned control, either some associated events have been set, a child timeout element has occurred, or a terminate element has occurred.
While_do. A while_do element evaluates its associated condition. If this condition evaluates to true, then the while_do element passes control to its child routing element. If this child routing element has returned control, the condition is re-evaluated, and so on. Thus, a while_do element takes care that its child routing element is performed as long as its associated condition holds.

Condition. A condition element also evaluates its associated condition. If this condition evaluates to true, then control is passed to all child true elements, otherwise control is passed to all child false elements. Every true and false element contains exactly one child routing element, and passes control to this child routing element. If this child routing element has returned control, then the true or false element returns control to the condition element. If all true of false elements (which depends on whether or not the condition holds) have returned control, the condition returns control. Thus, a condition element takes care that either all its child true elements are performed (if the condition holds) or all its child false elements are performed (if the condition does not hold).

As explained above, these routing elements of XRL are based on a thorough analysis of the workflow patterns supported by leading workflow management systems. Using these routing elements, we can model the Original net as shown by Figure 3.4 on page 46 and the Revised process as shown by Figure 3.3 on page 44. Figure 6.2 and Figure 6.3 show the resulting Original and Revised XRL process definitions. Note that, as our focus is on the control-flow perspective, both XRL process definitions only sketch the other perspectives. For example, the conditions are on a very abstract level. If our focus would have been on the data perspective too, then we would have introduced data elements and would have presented the conditions in more detail.

In the Original XRL process definition, after the client has connected, two threads of control run in parallel:

1. As long as the client is connected, he can either order a new item, or disconnect.
2. The company continually checks for pending orders. If there are pending orders, these are shipped. If there are no pending orders and if the client has disconnected, the company terminates the entire process.

In the Revised XRL process definition, the two parallel threads of control are more complicated:

1. As long as the company didn’t terminate the process, the user can be either connected or disconnected. If connected, he can order items or disconnect; if disconnected, he can reconnect. Also, after
ordering a first item, the client has to wait for the company to confirm.

2. The company continually checks for pending orders. If there are pending orders, a confirmation is sent to the client and the items are shipped. If there are no pending orders and if the client has disconnected, the entire transaction is archived and the entire process is terminated by the company.

6.3 WF-net semantics of XRL

The XRL DTD (or XRL Schema) only describes the syntax of XRL and does not specify its semantics. To provide operational semantics of the routing elements, we map each XRL routing element onto a net fragment. Such a mapping was given in [AK02]. However, as indicated earlier, this mapping does not necessarily yield WF-nets. Therefore, we have modified the mapping given in [AK02] such that XRL process definitions are mapped onto WF-nets. First, we discuss the general idea of the mapping and how we deal with events and terminates. Second, we present for the route element and for each routing element its net semantics. Third, we map both example XRL process definitions onto WF-nets.

```xml
<route name = “Original process”>
  <sequence>
    <task name = “connect” role = “client”/>
    <parallel_sync>
      <while_do condition = “client connected”>
        <choice>
          <task name = “order” role = “client”/>
          <task name = “disconnect” role = “client”/>
        </choice>
      </while_do>
      <while_do condition = “true”>
        <condition condition = “pending orders”>
          <true>
            <task name = “ship” role = “company”/>
          </true>
        </condition>
        <false>
          <terminate/>
        </false>
      </while_do>
    </parallel_sync>
  </sequence>
</route>
```

**FIGURE 6.2. The Original XRL process definition.**
6.3.1 Control passing and proper completion

In Section 6.2, we already mentioned that elements pass control to their child elements, and that elements return control to their parent element. The majority of routing elements needs only an interface that allows for this control passing and returning. For this reason, we propose to map every routing element onto some net fragment that passes control to its child net fragments and returns control to its parent net fragment. To enable the passing and returning of control between a child and its parent, we propose to insert two places between both their net fragments:

- one that allows the parent to pass control to the child, and

---

FIGURE 6.3. The Revised XRL process definition.
one that allows the child to return control to the parent.

However, according to the descriptions given in Section 6.2, there are three routing elements that can return control before all child elements have returned control, namely parallel_no_sync, parallel_part_sync, and parallel_part_sync_cancel. As a result, the top routing element might return control to the route element while, somewhere deep inside the instance, some routing elements still have control. Take, for instance, the Original route as shown in Figure 6.2 and suppose, for the sake of argument, that we replace the parallel_sync element by a parallel_no_sync element. Because the parallel_no_sync element can return control before both child while_do elements have returned control, the entire instance might reach completion before the while_do elements have returned control. Recall that the proper completion requirement of soundness states that the remainder of the entire WF-net must be empty when a token is put into the sink place, that is, a net fragment associated with any routing element has to be empty at that point. Therefore, before actually completing, we have be sure that all net fragments have returned control. For this reason, we introduce a third place, which allows a routing element to inform its parent that this routing element and all its descending elements have all returned control.

Figure 6.4 shows the template net fragment for any routing element, which contains five places and three transitions:

- A place $p_1$. A token in this place indicates that the parent has passed control to this routing element, that is, this routing element can be started.
- A place $p_2$. A token in this place indicates that this routing element has returned control to its parent.
- A place $p_3$. A token in this place indicates that this routing elements and all its descending elements have returned control, that is, except for this place their net fragments are all empty.
- A place $p_4$. A token in this place indicates that the routing element has control and is active.
• A place $p_5$. A token in this place indicates that the routing element has returned control to its parent and is waiting for all its descend-
ing elements to return control as well.
• A transition $t_1$. This transition starts the routing element.
• A transition $t_2$. This transition completes the routing element.
• A transition $t_3$. This transition signals to the parent that the routing
element and all its descendants have returned control.

Basically, every routing element contains the net fragment shown in
Figure 6.4. For this reason, when discussing the semantics of some
routing element, we use this basic template to model possible child
routing elements. As we are only interested in observing when tasks
are executed, we can label transitions $t_1$, $t_2$, and $t_3$ with $\tau$. Later on, we
see that a special transition corresponding to a task element is labeled
with the task’s name, which makes observing the task possible.

Until now, we have assumed that a routing element only needs to com-
municate with its parent element (to return control) and with its child
elements (to pass control). However, some elements clearly have an
instance-wide effect, namely task elements that set events, wait ele-
ments that test events, and terminate elements. Before presenting the
net fragments for the route element and all routing elements, we first
explain how we handle events and terminates.

6.3.2 Events

As mentioned above, events have an effect on the instance level, that
is, the route level, and not only on some local level deeply nested in
some subtree: If some task element in some subtree sets a certain
event, then some wait element in some other subtree might be affected.
For this reason, events are handled on the instance level: For every
event, we add a net fragment that handles that specific event. Such a
net fragment, called an event handler, interfaces only with the net frag-
ments associated with the route element (for starting and stopping the
event handler), task elements that have this event as a child element
(for having the event set), and both wait elements (for checking
whether the event has been set). An event handler contains four places
and three transitions and is shown by Figure 6.5.

Place $p_1$ holds the pending requests to have the event set, place $p_2$
holds the handled requests, place $p_3$ indicates that the event has been
set, and place $p_4$ indicates that the event has not been set. If the event
has not been set yet and if a request to set the event is pending, transi-
tion $t_2$ can set the event. If the event has been set, transition $t_1$ can han-
dle the request in a straightforward way. As a result, the first request to
have the event set, results in firing transition $t_2$ followed by firing tran-
WF-net semantics of XRL

6.3.3 Terminates

When a terminate element occurs, the entire instance, that is, the route element itself, is to be completed. This clearly has an effect on the instance level. However, it is hard to foresee all possible reachable states, and to add transitions such that from every reachable state we can reach the successful terminal state. Fortunately, a simple observation alleviates this problem: Only task elements and wait elements can delay completion for some time:

- a task element can delay completion because executing the corresponding activity takes time,
- a wait element can delay completion because some events are not set and a possible corresponding timeout takes time, and

Thus, if we prevent these routing elements from delaying completion, the instance automatically reaches completion! To actually prevent the these routing elements from delaying completion, if a terminate has occurred, we bypass the task an wait elements and we assume that the condition associated to the while_do element evaluates to false. Note that this solution assumes that active tasks are not preempted. The resulting terminate handler contains four places and three transitions, as is shown by Figure 6.6. Note that this net fragment is identical to the net fragment of an event handler. In fact, the occurrence of a terminate can be modeled by having a special terminate event set, which results

![Net fragment for an event handler.](image)

<table>
<thead>
<tr>
<th>handled request</th>
<th>event has been set</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
<td>p2</td>
</tr>
<tr>
<td>τ</td>
<td>t1</td>
</tr>
<tr>
<td>p3</td>
<td>t2</td>
</tr>
<tr>
<td>τ</td>
<td>p4</td>
</tr>
</tbody>
</table>

FIGURE 6.5. Net fragment for an event handler.

Note that all transitions have been labeled \( \tau \), as we are only interested in observing tasks, as mentioned above.
in the entire instance to move forwards to completion as fast as possible. A terminate element that needs to have this terminate event set, simply adds a token to the place p1 of the terminate handler, and waits until it can remove a token from the place p2 of the terminate handler.

We should not include the terminate handler if the XRL process definition does not contain a terminate element: If no terminate element is present, then the place p2 of the terminate handler would be a sink place. Therefore, the terminate handler is optional, and only present if terminate elements are present.

At this point, we have a general idea of the mapping and can introduce the net fragments of the route element and all routing elements.

6.3.4 Route element

Figure 6.7 shows the net fragment for the route element. Note that this figure contains references to the child routing element of the route element and to possible event (or terminate) handlers. As explained above, for sake of explanation, a template net fragment (see Figure 6.4) replaces the top routing element, which is the only child routing element of the route element. The shadow behind the event handler indicates that there may be multiple event handlers, but that, for sake of clarity, we only show one. Also note that in Figure 6.7 (and in many figures to come) we use similar identifiers for different objects. For example, in Figure 6.7, we have three transitions which seem to have t3 as identifier. However, these three transitions all belong to different net fragments: the first to the net fragment corresponding to the route element, the second to the net fragment corresponding to the child routing element, and the third to the net fragment belonging to the event handler. The combination of net fragment and identifier shown uniquely identifies the node, and is, therefore, used as formal node identifier. However, as these formal node identifiers tend to grow long, for sake of clarity, we only show short identifiers.
The net fragment for the route element contains three places and three transitions. Initially, place p1 (which corresponds to place i in Definition 3.9 on page 47) contains one token, indicating that the instance has not started yet. Transition t1 starts the instance and

1. passes control to the child routing element, and
2. starts every event handler and the optional terminate handler by adding a token to its place p4. (By default, no events are set and no terminate elements have occurred.)

If the child routing element has returned control, transition t2 initiates the completion phase of the instance. First, every event handler and the optional terminate is reset by its transition t3, which happens while place p3 is marked and because there are no other transitions enabled. Second, if all events and the optional terminate have been reset, and if all descending routing elements have returned control as well (which is indicated by a token in place p3 of the child routing element), transition t3 completes the instance and

1. removes the token from place p3,
2. disables every event handler and the optional terminate handler by removing the token from its place p4.

6.3.5 Terminate element

Figure 6.8 shows the net fragment for a terminate element. Recall that at the instance level a terminate handler is included whenever a terminate element is present, cf. Figure 6.6. Firing transition t1 requests the handler to have the terminate event set. Transition t2 can only fire if the
terminate event has been set. Thus, after a terminate element has returned control, the terminate event has been set.

6.3.6 Task element

Figure 6.9 shows the net fragment for a task element. Again, the shadow behind the event handler indicates that only one event handler is shown, but multiple may be present. Firing transitions t4 and t5 effectively bypasses the task if a terminate element has occurred. These transitions (and place p7) are only included if a terminate element exists in the XRL process definition, because they would be dead otherwise. Firing transitions t1 starts the task by offering the corresponding work item to some resources. Conceptually, transition t6
waits until the work item has been activated by some resource and until the corresponding activity has been completed by that resource. Thus, this transition effectively models the execution of the task. For this reason, this transition is labeled with the task’s name, which allows us to observe the execution of this task. Firing transition t6 requests the appropriate event handlers to have the associated events (could be none) set. Transition t2 can only fire if all associated events have been set. Thus, unless a terminate element has occurred, after a task element has returned control, all associated events have been set.

6.3.7 Sequence element

Figure 6.10 shows the net fragment for a sequence element. Firing transition t1 passes control to the first child routing element. As a result, this routing element can be started. When this routing element returns control, transition $t_{41}$ passes the control to place $p_1$ of the second child routing element, and so on (transition $t_4$ passes the control from the $i$-th child routing element to the $(i+1)$-th child routing element). Transition $t_2$ can only fire if the last routing element has returned control.
Note that the net fragment of a sequence element does not contain a place $p_4$ (see Figure 6.4), because such a place $p_4$ would be redundant: Adding such a place would not affect the behavior of the resulting WF-net. Several other routing elements exist for which place $p_4$ is redundant. These routing elements also do not contain this place.

6.3.8 Any_sequence element

Figure 6.11 shows the net fragment for an any_sequence element. Firing transition $t_1$ passes control to every child routing element. Furthermore, firing transition $t_4$ adds a token to the place $p_4$. The token in place $p_4$ guarantees the mutual exclusion of the child routing elements. A child routing element can only start if it can remove a token from place $p_4$, and when it starts, it takes the token. When the child routing element returns control, and its transition $t_2$ fires, the token is returned to place $p_4$. Transition $t_2$ can only fire if all child routing elements have returned control.

6.3.9 Choice element

Figure 6.12 shows the net fragment for a choice element. Firing transition $t_1$ passes control to every child routing element. The first child routing element that starts by firing its transition $t_1$, removes control from all other child routing elements. Because only one child routing element will be started, only one can return control. Therefore, transition $t_2$ can fire as soon as one of the child routing elements has returned control.

If a terminate element is present in the route, a task element will have an additional output transition $t_4$ for the place $p_1$ (see Figure 6.9). As a
result, if a terminate element is present in the route, and if a choice element contains a child task element, then arcs have to be added that connect the place p1 corresponding to any other child routing element of the choice element to the transition t4 corresponding to the child task element.

6.3.10 Parallel_sync element

Figure 6.13 shows the net fragment for a parallel_sync element. Firing transition t1 passes control to every child routing element. Transition t2 can only fire if all child routing elements have returned control.

6.3.11 Parallel_no_sync element

Figure 6.14 shows the net fragment for a parallel_no_sync element. Firing transition t1 passes control to every child routing element. Transition t2 can fire as soon as transition t1 has fired. Thus, the parallel_no_sync element can return control before any of its child routing elements have returned control. The returned controls of all child routing elements are redirected to transition t3.
6.3.12 Parallel_part_sync element

Figure 6.15 shows the fragment for a parallel_part_sync element, containing $n$ child routing elements of which $k$ need to have returned control before control can be returned. Firing transition $t_1$ passes control to every child routing element. Transition $t_2$ can only fire if at least $k$ child routing elements have returned control. The returned controls of the remaining $n-k$ child routing elements are redirected to transition $t_3$.

Note that place $p_4$ is not redundant for this routing element: If we would remove place $p_4$, transition $t_2$ could fire $\lfloor n/k \rfloor$ times. For example, if there are 10 child routing elements ($n = 10$) and the given
number is 3 \((k = 3)\), transition \(t2\) can fire 3 times, returning control 3 times, which is clearly an error.

Furthermore, note that this net fragment uses arc weights \((k\) and \(n\)), whereas the definition of nets (Definition 3.1 on page 39) does not allow for arc weights. For sake of completeness, we mention that a net with arc weights can be mapped onto a net without arc weights, by replacing every place with a number of places and transitions, while preserving liveness and boundedness. This is illustrated in Figure 6.16. The net on the left-hand side contains three arc weights, whereas in the net on the right-hand side these arc weights were removed by adding places and transitions. Note that we assume a strong notion of fairness for this mapping, because firing the additional transitions \(t_0, \ldots, t_3\) over and over again is clearly not acceptable: At some point in time, transition \(t1\) or transition \(t4\) is allowed to fire.

Recurrent re-entry of parallel_part_sync elements might lead to undesired behavior, as this might result in a parallel_part_sync element that,
for example, returns control after $k$ instances of its first child routing element have returned control. This can happen, for example, if a parallel_part_sync element descends some while_do element. To prevent this kind of undesired behavior, we prevent recurrent re-entry of the parallel_part_sync element by introducing place $p_7$. To enable the parallel_part_sync element, transition $t_1$ of the route element (see Figure 6.7) needs to add a token to this place, and transition $t_2$ of the route element needs to remove this token.

### 6.3.13 Parallel_part_sync_cancel element

Figure 6.17 shows the fragment for a parallel_part_sync_cancel element. This fragment is an extension of the net fragment for the parallel_part_sync element: Transitions $t_4$ have been added. After the parallel_part_sync_cancel has returned control, any child routing element that has not started yet can be withdrawn (cancelled) by firing the corresponding transition $t_4$.

### 6.3.14 Wait_all element

Figure 6.18 shows the net fragment for a wait_all element. Firing transition $t_1$ brings the wait_all element into a waiting state. Transition $t_2$ can only fire after any of the following has happened:

1. All associated events have been set, in which case transition $t_6$ has fired.
2. A terminate element has occurred, in which case transition $t_4$ has fired. Transition $t_4$ is only included if some terminate element exists in the XRL process definition, because it would be dead otherwise.
3. Some child timeout element has occurred, in which case the corresponding transition $t_7_i$ has fired, which started the optional child routing element of the child timeout element. If the child timeout element contains no child routing element, a template net fragment (see Figure 6.4) is included and started instead.

### 6.3.15 Wait_any element

Figure 6.19 shows the net fragment for a wait_any element. The net fragment for a wait_any element is identical to that of a wait_all element, except for the fact that transition $t_2$ can fire as soon as one event now has been set, in which case the corresponding transition $t_6_j$ has fired.

### 6.3.16 While_do element

Figure 6.20 shows the semantics of a while_do element. Firing transition $t_1$ results in one token in the place $p_2$ of the child routing element.
This token in place p2 indicates that all previous iterations have returned control. Furthermore, firing transition t1 also results in x tokens in place p3. Every token in place p3 corresponds to an available control. Firing transition t4 starts a new iteration for the child routing element, but this transition can only fire if all previous iterations have returned control (that is, if place p2 contains a token) and if a control is available (from place p3). As a result, at most x controls can exist in the child routing element and all its descending elements. As a result, if place p3 contains x tokens, then no other controls can exist in the child routing element and all its descending elements, and transition t3 can fire. Firing transition t3 removes the available controls from place p3, because they are not needed any more.

Note that x can be set to a very large value when enacting the entire XRL process definition, but for verification purposes x should be modest: The larger x is, the larger the state space of the entire resulting WF-net becomes.

6.3.17 Condition element

Figure 6.21 shows the net fragment for a condition element. After transition t1 has fired, the condition is evaluated. If the condition evaluated true, transition t4 fires, otherwise transition t7 fires. Transition t4
WF-net semantics of XRL


FIGURE 6.20. Net fragment for a while_do element.
passes control to the child routing element of every child true element, whereas transition $t7$ passes control to the child routing element of every child false element. After the child routing elements of all child true (false) elements have returned control, transition $t5$ ($t8$) fires, after which transition $t2$ can return control. Transition $t6$ ($t9$) fires if all descending routing elements of all child true (false) elements have returned control, after which transition $t3$ can fire.

6.3.18 Mapping onto a WF-net

At this point, all XRL routing elements can be mapped onto nets. By starting with the XRL route element and recursively mapping each child routing element onto its corresponding net fragment, one obtains a WF-net. It is clear that every routing element contains places $p1$, $p2$, and $p3$, and transitions $t1$, $t2$, and $t3$. To distinguish places and transitions from net fragments corresponding to different routing elements in running text, we subscript the places names with the routing element: If $r$ and $s$ are routing elements, then $p1_r$ identifies the place $p1$ in routing element $r$, and $p1_s$ identifies the place $p1$ in routing element $s$.

Theorem 6.1. Let $x$ be an XRL route element, let $r$ be the top routing element of $x$, and let routing element be mapped onto net $N = (P, T, F, l)$, as described above. Net $N$ is a WF-net.
Verification of XRL

Proof.

By induction on the routing elements.

1. For every routing element $r$, it is straightforward to check that every constituent node is either on some path from place $p_1$ to place $p_2$ or on some path from place $p_1$ to place $p_3$ (Figures 6.8 to 6.15 and 6.17 to 6.21).

2. All nodes corresponding to the route element or to any routing element are on some path from place $p_1_x$ to place $p_2_x$ (Result 1, Figure 6.7).

3. The terminate handler is only present in net $N$ if a terminate element is present in route $x$, thus all nodes from the terminate handler are on some path from place $p_1_x$ to place $p_2_x$ (Result 2, Figure 6.6, and Figure 6.7).

4. An event handler is only present when some task sets the corresponding event, thus all nodes from this event handler are on some path from place $p_1_x$ to place $p_2_x$ (Result 2, Figure 6.5, Figure 6.7, and Figure 6.9).

5. All nodes in net $N$ are on some path from place $p_1$ to place $p_2$ (Result 2, Result 3, and Result 4).

6. It is also straightforward to check that place $p_1$ is an input place and that place $p_2$ is an output place (Figures 6.5 to 6.15 and 6.17 to 6.21).

7. Net $N$ is a WF-net (Definition 3.9, Result 5, and Result 6, note that because all other nodes are on some path from place $p_1$ to place $p_2$, they cannot be source nodes or sink nodes).

For example, the Original XRL process definition (see Figure 6.2) is mapped onto a WF-net containing 84 places and 73 transitions, whereas the Revised XRL process definition (see Figure 6.3) is mapped onto a WF-net containing 140 places and 124 transitions. (Both WF-nets are not shown here because of their complexity.) As a result, we can check the important soundness property introduced in Chapter 3 using the diagnosis process presented in Chapter 4. However, without using this diagnosis process, we can already conclude some interesting properties related to soundness using only structural properties of XRL, as is demonstrated in the next section.

6.4 Verification of XRL

This section discusses the soundness of an XRL process definition, that is, the soundness of the WF-net the XRL route element is mapped onto. Recall that for soundness three requirements should hold. The first requirement states that the successful terminal state should be
reachable, that is, completion is possible. The second requirement states that completion is always proper, that is, no tokens are left after completion. The third requirement states that for every transition there is a way to fire it from the initial state. First, we show that any XRL process definition satisfies the proper completion requirement. Second, we show that only some routing elements can violate the other two requirements.

6.4.1 Proper completion

By using structural analysis, we show that completion of an XRL process definition is always proper. First, we show that the net corresponding to any routing element can be covered by place invariants if the routing element contains neither event elements nor terminate elements. Later on, we gradually add event elements and terminate routing elements.

Lemma 6.2.

Let \( r \) be a routing element that contains neither event elements nor terminate elements, and let routing element \( r \) be mapped onto net \( N = (P, T, F, I) \), as described in Section 6.3. There exist two place invariants \( v_r \) and \( w_r \) of net \( N \) such that:

1. \( v_r(p_{1,r}) = v_r(p_{2,r}) \),
2. \( w_r(p_{1,r}) = w_r(p_{3,r}) \), and
3. for all \( p \in P : (v_r + w_r)(p) > 0 \).

Proof.

By induction on the routing elements. For sake of simplicity, we assume that the place invariants of the child routing elements are recalibrated such that the places \( p_2 \) and \( p_3 \) have weight 1. As a result, some weights might be fractions instead of natural numbers, but for the proof this is irrelevant: If we would multiply all weights by the product of their denominators, then all weights would be natural numbers.

1. A task element \( r \) (see Figure 6.9):

\[
\begin{align*}
\cdot v_r &= p_{1,r} + p_{2,r} + p_{4,r} + p_{6,r} + p_{7,r} \\
\cdot w_r &= p_{1,r} + p_{3,r} + p_{4,r} + p_{5,r} + p_{6,r} + p_{7,r}
\end{align*}
\]

2. A sequence element \( r \) (see Figure 6.10), with \( n \) child routing elements \( r_1, \ldots, r_n \):

\[
\begin{align*}
\cdot v_r &= p_{1,r} + p_{2,r} + \sum_{i=1}^{n} v_{r_i} \\
\cdot w_r &= (n + 1)(p_{1,r} + p_{3,r}) + p_{5,r} + \sum_{i=1}^{n} (n + 1 - i)v_{r_i} + w_{r_i}
\end{align*}
\]
3. An any_sequence element $r$ (see Figure 6.11), with $n$ child routing elements $r_1, \ldots, r_n$:

\[
\nu_r = (n + 1)(p1_r + p2_r) + p4_r + \left( \sum_{i=1}^{n} v_{r_i} + p4_{r_i} \right)
\]

\[
w_r = (2n + 1)(p1_r + p3_r) + p4_r + (n + 1)p5_r + \left( \sum_{i=1}^{n} v_{r_i} + w_{r_i} + p4_{r_i} \right)
\]

4. A choice element $r$ (see Figure 6.12), with $n$ child routing elements $r_1, \ldots, r_n$:

\[
\nu_r = n(p1_r + p2_r + p6_r) + \left( \sum_{i=1}^{n} n(v_{r_i} - (n - 1)p1_{r_i}) \right)
\]

\[
w_r = 2n(p1_r + p3_r) + n(p5_r + p6_r + p7_r) + \left( \sum_{i=1}^{n} n(v_{r_i} + w_{r_i}) - (n - 1)p1_{r_i} \right)
\]

5. A parallel_sync element $r$ (see Figure 6.13), with $n$ child routing elements $r_1, \ldots, r_n$:

\[
\nu_r = n(p1_r + p2_r) + \left( \sum_{i=1}^{n} v_{r_i} \right)
\]

\[
w_r = 2n(p1_r + p3_r) + np5_r + \left( \sum_{i=1}^{n} v_{r_i} + w_{r_i} \right)
\]

6. A parallel_no_sync element $r$ (see Figure 6.14), with $n$ child routing elements $r_1, \ldots, r_n$:

\[
\nu_r = p1_r + p2_r + p4_r
\]

\[
w_r = (2n + 1)(p1_r + p3_r) + p4_r + p5_r + \left( \sum_{i=1}^{n} v_{r_i} + w_{r_i} \right)
\]
7. A parallel_part_sync element \( r \) (see Figure 6.15), with \( n \) child routing elements \( r_1, \ldots, r_n \):
   - \( v_r = p_1 + p_2 + p_4 \)
   - \( w_r = 2n(p_1 + p_3) + p_4 + (k + 1)p_5 + p_6 + p_7 + \left( \sum_{i=1}^{n} v_{r_i} + w_{r_i} \right) \)

8. A parallel_part_sync_cancel element \( r \) (see Figure 6.17), with \( n \) child routing elements \( r_1, \ldots, r_n \): See Result 7 (identical to the place invariants of the parallel_part_sync element).

9. A wait_all element \( r \) (see Figure 6.18), with \( n \) timeout elements that each have a one child routing element \( r_i \):
   - \( v_r = p_1 + p_2 + p_6 + p_7 + \left( \sum_{i=1}^{n} v_{r_i} \right) \)
   - \( w_r = 2(p_1 + p_3 + p_6) + p_5 + p_7 + p_8 + \left( \sum_{i=1}^{n} v_{r_i} + w_{r_i} \right) \)

10. A wait_any element \( r \) (see Figure 6.19), with \( n \) timeout elements that each have a one child routing element \( r_i \): See Result 9 (identical to the place invariants of the wait_all element).

11. A while_do element \( r \) (see Figure 6.20), with one child routing element \( s \):
   - \( v_r = p_1 + p_2 + p_4 \)
   - \( w_r = (x + 1)(p_1 + p_3) + p_5 + v_s + w_s \)

12. A condition element \( r \) (see Figure 6.21), with \( n \) child true elements that have each one child routing element \( t_i \), and \( m \) child false elements that have each one child routing element \( f_i \):
   - \( v_r = (n + m + 1)(p_1 + p_2 + p_6 + p_7) + (m + 1)p_9 \)
   + \( (n + 1)p_{11} + \left( \sum_{i=1}^{n} v_{t_i} \right) + \left( \sum_{i=1}^{m} v_{f_i} \right) \)
Verification of XRL

Second, we show that the WF-net corresponding to an XRL process definition can be covered by a place invariant such that its source place \( p_1 \) and sink place \( p_2 \) have identical weights.

\[
\begin{align*}
\text{Lemma 6.3.} & \quad \text{Let } x \text{ be an XRL route element that contains neither terminate elements nor event elements, let } r \text{ be the child routing element of } x, \text{ and let route element } x \text{ be mapped onto WF-net } N = (P, T, F, l), \text{ as described in Section 6.3. There exists a place invariant } w_x \text{ of net } N \text{ such that:} \\
1. & \quad w_x(p_1_x) = w_x(p_2_x) \text{ and} \\
2. & \quad \text{for all } p \in P : w_x(p) > 0.
\end{align*}
\]

Proof. By construction.

1. \( w_x(p_1_x) = w_x(p_2_x) \) (Lemma 6.2).

Third, we show that the WF-net corresponding to an XRL route element that contains one event, can be covered by such a place invariant.

\[
\begin{align*}
\text{Lemma 6.4.} & \quad \text{Let } x \text{ be an XRL route element that contains one event element named } e, \text{ but that contains no terminate elements, and let route element } x \text{ be mapped onto WF-net } N = (P, T, F, l), \text{ as described in Section 6.3. There exists a place invariant } w_{x,e} \text{ such that:} \\
1. & \quad w_{x,e}(p_1_x) = w_{x,e}(p_2_x) \text{ and} \\
2. & \quad \text{for all } p \in P : w_{x,e}(p) > 0.
\end{align*}
\]

Proof. By construction.

1. The place invariant \( w_x \) covers all places that are not related to the event handler (Lemma 6.3).

2. The place invariant \( p_1_x + p_2_x + p_3_e + p_4_e \) covers the places \( p_3_e \) and \( p_4_e \), and contains places \( p_1_x \) and \( p_2_x \) with identical weights.
3. Let \( w_e \) be equal to \( w_x \), except for the tasks where event \( e \) is set. For such a task \( r \), the occurrence of place \( p_6 \) is replaced by the expression \( p_1 \) + \( p_2 \). It is straightforward to check that \( w_e \) is a place invariant covering places \( p_1 \) and \( p_2 \), and containing places \( p_1 \) and \( p_2 \) with identical weights.

4. The place invariant \( w_{x,e} = w_x + w_e + p_1 + p_2 + p_3 \) covers all places and contains places \( p_1 \) and \( p_2 \) with identical weights (Result 1, Result 2, Result 3).}

Fourth, we show that the WF-net corresponding to an XRL route element that does not contain terminate elements, can be covered by such a place invariant.

**Lemma 6.5.**

*Let \( x \) be an XRL route element that does not contain terminate elements, let \( E \) be the set of events in route element \( x \), and let route element \( x \) be mapped onto WF-net \( N = (P, T, F, l) \), as described in Section 6.3. Then there exists a place invariant \( w_{x,E} \) such that:

1. \( w_{x,E}(p_1) = w_{x,E}(p_2) \) and
2. for all \( p \in P : w_{x,E}(p) > 0 \).*

**Proof.**

By construction.

1. \( w_{x,E} = \sum_{e \in E} w_{x,e} \) (Lemma 6.4).

Last, we show that the WF-net corresponding to an arbitrary XRL route element can be covered by such a place invariant.

**Lemma 6.6.**

*Let \( x \) be an XRL route element, let \( E \) be the set of events in route element \( x \), and let route element \( x \) be mapped onto the WF-net \( N = (P, T, F, l) \), as described in Section 6.3. There exists a place invariant \( w_{x,E,term} \) such that:

1. \( w(p_1) = w(p_2) \) and
2. for all \( p \in P : w(p) > 0 \).*

**Proof.**

By construction.

1. Lemma 6.2 can be extended with terminate elements in a straightforward way: For a terminate element \( r \) (see Figure 6.6):
   - \( v_r = p_1 + p_2 + p_4 \)
   - \( w_r = p_1 + p_3 + p_4 + p_5 \)
2. The extended place invariant $w_{x,E}$ covers all places that are not related to the terminate handler (Result 1 and Lemma 6.5).

3. The place invariant $p_1 + p_2 + p_3_{\text{term}} + p_4_{\text{term}}$ covers the places $p_3_{\text{term}}$ and $p_4_{\text{term}}$, and contains places $p_1$ and $p_2$ with identical weights.

4. Let $w_{\text{term}}$ be equal to $w_x$, except for the terminate elements. For every terminate element $t$, the occurrence of place $p_4$ is replaced by the expression $p_1 + p_2$. It is straightforward to check that $w_{\text{term}}$ is a place invariant containing places $p_1$ and $p_2$ with identical weights.

5. The place invariant $w = w_{x,E} + w_{\text{term}} + p_1 + p_2 + p_3_{\text{term}} + p_4_{\text{term}}$ covers all places and contains places $p_1$ and $p_2$ with identical weights (Result 2, Result 3, Result 4).

As a result, we may conclude that the WF-net corresponding to an XRL route satisfies the proper-completion property of soundness.

**Theorem 6.7.**

Let $x$ be an XRL route element, let $E$ be the set of events in route element $x$, and let route element $x$ be mapped onto the WF-net $N = (P, T, F, I)$, as described in Section 6.3. WF-net $N$ satisfies the proper-completion requirement of soundness (see Definition 3.39 (Soundness)).

**Proof.**

1. A place invariant $w$ exists in net $N$ such that $w(p_1) = w(p_2)$ and for all $p \in P$: $w(p) > 0$ (Lemma 6.6).

2. Invariant $w$ is also a place invariant of the short-circuited net $\varphi N$ (Result 1, Definition 3.11 (Short-circuited WF-net), and Definition 3.24 (Place invariant)).

3. The system $(\varphi N, [p_1], [p_2])$ is bounded (Result 2, see for instance Theorem 2.31 from [DE95]).

4. The WF-net $N$ satisfies the proper-completion requirement of soundness (Result 3 and Theorem 4.5).

We have shown, using only structural analysis techniques, that completion of an XRL process definition (or an XRL route element) is always proper. Next, we show that only some routing elements can violate the other two soundness requirements.

### 6.4.2 Option to complete and no dead tasks

First, we show that the option-to-complete requirement can only be violated by a wait element that does not contain a timeout element.
Theorem 6.8. Let $x$ be an XRL route element and let route element $x$ be mapped onto WF-net $N$, as described in Section 6.3. If every wait element in route element $x$ contains a timeout element, then net $N$ satisfies the option-to-complete requirement of soundness (see Definition 3.39 (Soundness)).

Proof. By induction on the routing elements.

1. It is straightforward to check that only a wait element that does not contain a timeout element can delay completion indefinitely (Figures 6.5 to 6.15 and 6.17 to 6.21). Note that due to our fairness assumptions, and because we abstract from data, a while_do element cannot delay completion indefinitely.

Second, we show that the no-dead-transitions requirement can only be violated by a wait element or by a terminate element.

Theorem 6.9. Let $x$ be an XRL route element and let route element $x$ be mapped onto WF-net $N$, as described in Section 6.3. If route element $x$ contains no wait or terminate elements, then net $N$ satisfies the no-dead-transitions requirement of soundness (see Definition 3.39 (Soundness)).

Proof. This proof uses the fact that, because route element $x$ contains no wait element, the option-to-complete requirement is satisfied. As a result, a transition on some path can only be dead if an alternative path exists that can be enforced.

1. Route element $x$ contains no wait elements (presupposition).
2. Route element $x$ satisfies the option-to-complete requirement (Result 1 and Theorem 6.8).
3. Only the wait elements contain alternative paths that can be enforced (Figures 6.5 to 6.15 and 6.17 to 6.21). Note that alternative paths in the condition and while_do elements cannot be enforced, as we abstracted from data, while the alternative paths in the task elements cannot be enforced because terminate elements are absent.

Corollary 6.10. Let $x$ be an XRL route element and let route element $x$ be mapped onto WF-net $N$, as described in Section 6.3. There are two possible causes for net $N$ to be not sound:

1. a wait_all or wait_any element delays completion indefinitely or contains dead transitions, or
2. a terminate element is present which results in dead transitions.

A terminate element can cause dead transitions in two ways:
1. If some routing element is always preceded by a terminate element, then all transitions corresponding to that routing element are dead.

2. If a task element always precedes every terminate element, then the transitions $t_4$ and $t_5$ corresponding to that task element are dead. Likewise, if a wait element always precedes every terminate element, then the transition $t_4$ corresponding to that wait element is dead.

However, almost by definition, a terminate element cannot cause the short-circuiting transition to be dead. As a result, a transition in the WF-net is either dead or live. A wait element can cause the short-circuiting transition to be dead, because such a construct may delay completion indefinitely. As a result, some other transitions (for example, the transition $t_1$ corresponding to the route element) will not be live, but it will not be dead either. Thus, if some transitions are dead while the others are all live, then this can only be caused by a terminate element; if some transition are not live and not dead, then this can only be caused by a wait element.

Both the Original XRL process definition (see Figure 6.2) as the Revised XRL process definition Figure 6.3) do not contain wait elements, but both contain a terminate element. Thus, some transitions might be dead. In both route elements, the task element named connect precedes the terminate element. As a result, the corresponding transitions $t_4$ and $t_5$ are dead. Obviously, this is not really an error. We could try to tackle this problem by trying to decide which task elements precede every terminate element. However, due to the presence of events, the precedence relations may be very complex. For this reason, we decided to use another solution: We change the XRL route element such that exactly these formerly dead transitions are not dead, by introducing a new terminate element that can precede any other routing element:

- First, we ‘demote’ the top routing element to a child routing element of a new top parallel_no_sync element.
- Second, we add a terminate element as an additional child element to this new parallel_no_sync element.

As a result, no task elements need to precede this terminate element. Figure 6.22 shows the Original XRL process definition (see Figure 6.2) after these changes were made. The corresponding WF-net contains 94 places and 79 transitions, the WF-net corresponding to the changed Implementation XRL process definition (see Figure 6.3) contains 150 places and 130 transitions.
6.4.3 Reduction techniques

Using the mapping from XRL process definitions as described in Section 6.3, the theory and tools for WF-nets can be deployed in a straightforward manner. This allows us to use the diagnosis process as described in Chapter 4 for verifying the correctness of an XRL route using the soundness property. However, XRL process definitions with a lot of parallelism tend to have a large state space, which might complicate verification from a computational point of view. Therefore, we propose a verification procedure that consists of two optimization steps. In the first step, the XSL mapping, which maps the XRL process definition onto a WF-net, reduces the WF-net by using structural properties of XRL. In the second step, the resulting WF-net is reduced even further by applying the liveness and boundedness preserving reduction rules as described in Chapter 4.

The first step is the reduction by the XSL mapping based on structural properties of XRL. This reduction is threefold:

```xml
<route name = “Specification process”>
  <parallel_no_sync>
    <terminate/>
    <sequence>
      <task name = “connect” role = “client”/>
      <while_do condition = “client_connected”>
        <choice>
          <task name = “order” role = “client”/>
          <task name = “disconnect” role = “client”/>
        </choice>
      </while_do>
      <while_do condition = “true”>
        <condition condition = “pending_orders”>
          <true><task name = “ship” role = “company”/></true>
          <false><terminate/></false>
        </condition>
      </while_do>
    </sequence>
  </parallel_no_sync>
</route>
```

FIGURE 6.22. The Original XRL process definition to verify.
1. Figures 6.8 to 6.15 and 6.17 to 6.21 show a place named p3 to accommodate the situation where a routing element can return control before all its descending elements have returned control. This situation can only occur if the routing element ascends some parallel_no_sync, parallel_part_sync, or parallel_part_sync_cancel element. In all other cases, there is no need to include these places p3. For instance, assuming the while_do element in Figure 6.20 has no need for place p3 allows us to remove its places p5 and p3 and the transition t3. As a result, its child routing element also does not need a place p3, and so forth.

2. If a task element contains no child event elements, transition t6 and place p6 in that task element can be replaced by a single arc from place p4 to transition t2.

3. We can apply the result presented in [AVK01a]: a routing element that does not ascend any event, wait, or terminate element cannot violate soundness and can therefore be replaced by the template net fragment shown in Figure 6.4. Note that this is consistent with the results in Section 6.4.2.

When the above-mentioned rules are applied, the XRL route shown in Figure 6.22 is transformed into a WF-net that contains only 43 places and 39 transitions. Compared to the original WF-net (94 places and 79 transitions), the reduced WF-net is considerably smaller and less complex. Note that several routing elements can be abstracted from, and that the resulting WF-net need only a few p3 places (because the parallel_no_sync elements are at a high level in the tree).

The second step is the reduction of the resulting WF-net based on well-known liveness and boundedness preserving reduction rules (see Section 4.4). Fragments of various routing elements are connected by transitions. This introduces a lot of transitions that are not relevant for the verification but introduce transient states. These and other parts of the WF-net can be reduced without losing information relevant for the verification. In Chapter 3, it was pointed out that soundness corresponds to liveness and boundedness [Aal98a]. This allows us to apply the well-known liveness and boundedness preserving reduction rules for Petri nets, which were presented in Section 4.4. These rules are shown in Figure 4.11 on page 93. Note that not all rules are relevant for reducing a WF-net derived from an XRL route: For instance the last rule will not be applied, because the only marked place in the WF net has no input arcs. After these reduction rules are applied to the reduced WF-net mentioned under Step 1, the resulting WF-net contains only 25 places and 21 transitions and is shown in Figure 6.23. For sake of readability, in this figure, the corresponding routing elements are shown by rounded boxes. Table 1 shows how the reductions affect the size (in number of places and transitions) of the WF-nets corresponding to the
FIGURE 6.23. The reduced WF-net corresponding to the XRL route from Figure 6.22.
Verification of XRL

Verification of XRL specification route and the implementation route, after a parallel_no_sync element and a terminate element have been added, as described in Section 6.4.2. The soundness results from Section 6.4 still hold after applying these reduction rules. It is straightforward to check that the transition t6 in a wait_all element or a transition t6 in a wait_any element will not be reduced, and that any terminate element will also not be reduced. Note that we do not apply the six WF-net-based reduction rules on the short-circuited net, but on the original WF-net. As a result, the short-circuited transition will still be present after the WF-net has been reduced. As mentioned in Section 6.4, this transition can be very useful when diagnosing the net.

### 6.4.4 Verification procedure

We propose a verification procedure that consists of three steps. In the first step, the XRL route is transformed into a WF-net, taking into account the three reduction rules based on the structure of XRL. In the second step, the resulting WF-net can be reduced even further using the six reduction rules based on the structure of the WF-net. In the third step, we use the diagnosis process from Chapter 4 to verify and diagnose the reduced WF-net.

Using standard Petri-net-based analysis tools, or dedicated tools such as Woflan, it is easy to show that the WF-net shown in Figure 6.23 is sound. Therefore, the XRL route shown in Figure 6.2 is correct, that is, free of deadlocks, livelocks and other anomalies. Likewise, we can show that the Implementation route of Figure 6.3 is correct.

<table>
<thead>
<tr>
<th>WF-net</th>
<th>Places</th>
<th>Transitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original net</td>
<td>94</td>
<td>79</td>
</tr>
<tr>
<td>— after XRL-based reduction</td>
<td>43</td>
<td>39</td>
</tr>
<tr>
<td>— after XRL-based and net-based reductions</td>
<td>25</td>
<td>21</td>
</tr>
<tr>
<td>Revised net</td>
<td>150</td>
<td>130</td>
</tr>
<tr>
<td>— after XRL-based reduction</td>
<td>49</td>
<td>46</td>
</tr>
<tr>
<td>— after XRL-based and net-based reductions</td>
<td>27</td>
<td>24</td>
</tr>
</tbody>
</table>

**TABLE 6.1. Results of XRL-based and Petri-net-based reductions.**

Specification route and the Implementation route, after a parallel_no_sync element and a terminate element have been added, as described in Section 6.4.2.

The soundness results from Section 6.4 still hold after applying these reduction rules. It is straightforward to check that the transition t6 in a wait_all element or a transition t6 in a wait_any element will not be reduced, and that any terminate element will also not be reduced. Note that we do not apply the six WF-net-based reduction rules on the short-circuited net, but on the original WF-net. As a result, the short-circuited transition will still be present after the WF-net has been reduced. As mentioned in Section 6.4, this transition can be very useful when diagnosing the net.

### 6.4.4 Verification procedure

We propose a verification procedure that consists of three steps. In the first step, the XRL route is transformed into a WF-net, taking into account the three reduction rules based on the structure of XRL. In the second step, the resulting WF-net can be reduced even further using the six reduction rules based on the structure of the WF-net. In the third step, we use the diagnosis process from Chapter 4 to verify and diagnose the reduced WF-net.

Using standard Petri-net-based analysis tools, or dedicated tools such as Woflan, it is easy to show that the WF-net shown in Figure 6.23 is sound. Therefore, the XRL route shown in Figure 6.2 is correct, that is, free of deadlocks, livelocks and other anomalies. Likewise, we can show that the Implementation route of Figure 6.3 is correct.

<table>
<thead>
<tr>
<th>WF-net</th>
<th>Places</th>
<th>Transitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original net</td>
<td>94</td>
<td>79</td>
</tr>
<tr>
<td>— after XRL-based reduction</td>
<td>43</td>
<td>39</td>
</tr>
<tr>
<td>— after XRL-based and net-based reductions</td>
<td>25</td>
<td>21</td>
</tr>
<tr>
<td>Revised net</td>
<td>150</td>
<td>130</td>
</tr>
<tr>
<td>— after XRL-based reduction</td>
<td>49</td>
<td>46</td>
</tr>
<tr>
<td>— after XRL-based and net-based reductions</td>
<td>27</td>
<td>24</td>
</tr>
</tbody>
</table>

**TABLE 6.1. Results of XRL-based and Petri-net-based reductions.**
6.5 Conclusions

XRL embeds the most relevant patterns found in the workflow management field today [AHKB00]. As a result, its expressive power far exceeds the expressive powers of a lot of existing workflow management systems. Therefore, if we can map XRL onto WF-nets, then we can map workflow process definitions of a lot of workflow management systems onto WF-nets. Furthermore, XRL is instance-based and extendible. As a result, at any time, routing elements may be added to or removed from an XRL process definition (that is, from a ‘live’ case), including newly conceived routing elements. Therefore, verification of an XRL process definition is of the utmost importance.

We have introduced a mapping from XRL to WF-nets. As such, we have shown that it is possible to automatically map any of the most relevant patterns found in the workflow management field today onto a WF-net. Therefore, we conclude that, for a lot of existing workflow management systems, workflow process definitions can be mapped onto WF-nets. As a result, we can decide soundness of many workflow process definitions by deciding soundness on its corresponding WF-net.

We have also shown that the selected (most relevant) workflow patterns satisfy the proper completion requirement, and that the other two requirement can only be violated by some of the selected patterns: The option-to-complete requirement can only be violated by a pattern that corresponds to a wait element, whereas the no-dead-tasks requirement can only be violated by a pattern that corresponds to a wait or terminate element. As a result, verification of an XRL process definition can focus on these wait and terminate elements, and can abstract from parts of the XRL process definition that are not related to wait or terminate elements. We have specified which parts can be abstracted from (that is, can be reduced). Furthermore, we have demonstrated that the liveliness and boundedness preserving reduction rules (see Section 4.4.2 on page 94) can be applied successfully on the mapped XRL process definition.
The previous chapter shows that, in general, the control flow of workflow processes can be mapped onto WF-nets, using the eXchangeable Routing Language (XRL) as proof-of-concept. This chapter presents mappings from process definitions specified in some WFMSs in particular, namely IBM MQSeries Workflow, Staffware, and COSA Workflow, and of process definitions specified using the BPR tool Protos, onto WF-nets. Thus, whereas the previous chapter shows that mapping workflow process definitions should be possible, this chapter shows that for the above-mentioned systems this is indeed possible. Furthermore, this chapter shows what effect the peculiarities of these systems have on the diagnosis process as presented in Chapter 4.

We selected the Staffware WFMS because it is one of the leading WFMSs in the world: In 1998, Gartner estimated that Staffware had a 25 percent share of the global market [AHKB03, Cas98]. Furthermore, Staffware is interesting from a verification point-of-view, because the enactment of a Staffware workflow process definition might behave different than anticipated by the designer: Staffware process definition seem to have an intuitive semantics, but some pitfalls exist that the designer needs to be aware of.

We selected the COSA Workflow WFMS because it is Petri-net-based, which possibly makes the entire diagnosis process as described in Chapter 4 valuable, and because it is one of the leading WFMSs in The Netherlands.

We selected the IBM MQSeries Workflow WMFS because of its peculiar semantics. In [Kie02, KHA03], a classification for workflow process definitions of existing WFMSs is presented. According to this
Mappings

182

classification, the following classes of workflow process definitions can be identified:

- **Standard workflow process definitions**, which basically correspond to nets. According to [KHA03], this class is “what would appear to be the most “natural” interpretation of the WfMC definitions” [KHA03]. Petri-net-based workflow process definitions typically belong to this class.

- **Safe workflow process definitions**, which basically correspond to nets except for the fact that multiple tokens in a place get merged into one. As a result, a place contains at most one token.

- **Structured workflow process definitions** [Kie02], which basically correspond to nets with properly matching joins and splits.

- **Synchronizing workflow process definitions**, which basically correspond to nets with only AND-joins/splits in combination with false/true tokens. A task only has a corresponding work item if the corresponding token is a true token. This is an altogether different approach that allows for support of some advanced workflow join patterns [AHKB03] that are difficult to support by standard, safe, or structured workflow process definitions.

Thus, the first three classes are (more or less) related, but the class of synchronizing workflow process definitions is fundamentally different. IBM MQSeries Workflow belongs to this class of synchronizing workflow process definitions. For sake of completeness, we mention that both Staffware and COSA Workflow belong to the class of safe workflow process definitions. Note that although COSA Workflow is based on Petri nets, it does not allow multiple concurrent instances of activities, and hence corresponds to the class of safe workflow process definitions.

We selected the BPR tool Protos because this tool is used in one of our courses that also focuses on verification issues. As a part of this course, in 2002, eleven groups of three students had to model, using Protos, a sound workflow process definition for some (imaginary) process. Next, they had to introduce three errors and pass their erroneous workflow process definition to two other groups, which had to find and correct the errors introduced. Using Woflan, nine groups were able to correct both erroneous workflow process definitions handed to them, while the remaining two groups were able to correct only one (these groups delivered two sound workflow process definitions, but the corrections made in one of these definitions were unacceptable.)

The remainder of this chapter is organized as follows. First, we investigate how MQSeries Workflow processes can be mapped onto WF-nets and what effect this mapping has on soundness and its diagnosis process (see Figure 4.14 on page 97). Second, third, and fourth, we inves-
MQSeries Workflow

MQSeries Workflow [IBM01b], nowadays known as WebSphere MQ Workflow, is the successor of FlowMark, the workflow management systems of IBM. The following description of MQSeries Workflow is based on [LR99, Köh03, IBM01b].

7.1 MQSeries Workflow

7.1.1 Processes

An MQSeries Workflow process definition contains processes, activities, and connectors. An activity and a connector can be either true or false.

Process. A process is a unified acyclic graph, with activities as nodes and connectors as edges, where a graph is unified if and only if between every two nodes there is at most one edge.

Activity. There are three kinds of activities:

1. A \textit{program} activity models a task.
2. A \textit{process} activity refers to a process and models a subprocess.
3. A \textit{block} activity also refers to a process and models an iterative subprocess (a loop).

Every activity has a \textit{join condition}. A join condition can either be an “all” condition or an “at-least-one” condition. In Chapter 5 of [LR99], some advanced join conditions are mentioned, but as these are not yet implemented in MQ Workflow, thus, we do not address these advanced join conditions. An “all” condition acts as an AND-join (all incoming connectors need to be true), whereas an “at-least-one” condition acts as an OR-join (at least one of the incoming connectors needs to be true). An activity is true if and only if its join condition evaluates to true. A true activity gets executed, whereas a false activity gets skipped. An activity also has an \textit{exit condition}, but this condition is only relevant for block activities: A block activity iterates its subprocess as long as the exit condition evaluates to false. An activity without incoming connectors is called a \textit{start} activity, and an activity without outgoing connectors is called an \textit{end} activity.

Connector. There are two kinds of connectors: \textit{control} connectors and \textit{default} connectors [Köh03, IBM01b]. A control connector can have an associated condition. For a false activity, every outgoing connector is
Mappings

false. For a true activity, this depends on the (optional) associated condition:

- every outgoing control connector that has no associated condition is true,
- every outgoing control connector that has an associated condition is true if and only if the associated condition evaluates to true, and
- every outgoing default connector is true if and only if every outgoing control connector of the activity is false.

7.1.2 Sample processes

Figure 7.1 shows an MQSeries process definition for the Original process definition (see Section 3.2), whereas Figure 7.2 shows an MQSeries process definition for the Revised process definition (see also Section 3.2).

In the Original MQSeries process definition, first the client can connect, and a loop (modeled by a block activity) runs until the client has disconnected. During every iteration of that loop, the client first has to decide whether he wants to order an additional item. If so, the client can order the item, else, the client is disconnected. If the client ordered
an item, two things happen in parallel: the company can ship, or a second loop is started which runs until the company has shipped. During this second loop, the client has to decide whether he wants to order an additional item. If so, the item will be added to the shipment, otherwise, the client is disconnected and the process will end.

In the Revised MQSeries process definition, more or less the same things happen in the same order, with the following exceptions:

- The company now first has to confirm before it can ship and before the client can order additional items.
- After the client is disconnected, he can decide to reconnect. If so, the first loop will continue and the client can order items for an additional shipment, otherwise, the company can archive the entire transaction and the process will end.

### 7.1.3 Mapping

In [Kie02, KHA03], a semantics is given for a class of workflow models with a synchronizing control-flow strategy. However, although the authors mention that “With slight modifications, this view was suc-

![Diagram](image-url)
cessfully implemented by IBM’s MQSeries Workflow”, the proposed semantics is not suitable for our purposes for the following reasons:

- Process activities and block activities are not considered.
- Only explicit XOR-splits and AND-splits are considered, while in fact [AHKB03] MQSeries Workflow uses implicit OR-splits. To model OR-splits in the same way XOR-splits and AND-splits are modelled in [KHA03], an explosion in the number of transitions would be required.
- The semantics for every program activity contains false places and false transitions, while for some activities these places and activities are not needed. Even worse, because such an activity might not be needed, it will be dead and, thus, hinder the verification. A straightforward example for an activity that does not need these places and transitions is a start activity.

For these reasons, we do not use the semantics as given in [KHA03] to verify MQSeries Workflow processes. However, we do use the idea of true tokens and false tokens.

Similar to XRL in the previous chapter, to every activity, connector, and process, a net fragment is associated. Depending on the situation at hand, some places and transitions need to be removed from the fragment. The net obtained by combining the resulting net fragments is a WF-net if the processes are connected.

**Activity.** Figure 7.3 shows the net template for an activity with \( n \) incoming connectors. (Note that, like in the previous chapter, we use
MQSeries Workflow

(dashed) regions to tell nodes with identical names apart, because regions seem much more elegant than subscripts.) Later on, we see that an incoming true connector puts a token into place p1, and that an incoming false connector puts a token into place p2. Places p1 and p2 model the evaluation of the join condition of the activity: If place p1 contains \( n \) tokens, then the join condition evaluates to true; if the place p2 contains \( n \) tokens, then the join condition evaluates to false. Thus, if all incoming connectors are true, the join condition evaluates to true, and if all incoming connectors are false, the join condition evaluates to false. Transitions t7 and t8 take care of the possibility that some incoming connectors are true while others are false:

- In case of an “at-least-one” join condition, if any of the incoming connectors is true, the join condition evaluates to true. This is modeled by transition t7: If place p1 contains a token, then all tokens from place p2 can be transferred to place p1 by this transition, resulting in \( n \) tokens in place p1.
- In case of an “all” join condition, if any of the incoming connectors is false, the join condition evaluates to false. This is modeled by transition t8: If place p2 contains a token, then all tokens from place p1 can be transferred to place p2 by this transition, resulting in \( n \) tokens in place p2.

Note that transitions t7 and t8 need to be removed if only one incoming transition exists, that transition t7 needs to be removed in case of an “all” join condition, and that transition t8 needs to be removed in case of an “at-least-one” join condition.

If the join condition has evaluated to false, and place p2 contains \( n \) tokens, then transition t2 skips this activity. Later on, we see that this transition starts all outgoing connectors, and that every these outgoing connector is false. In some cases, a join condition cannot evaluate to false. In these cases, transition t2 needs to be removed from the net fragment.

If the join condition has evaluated to true, and place p1 contains \( n \) tokens, then transition t3 starts execution of this activity. Transitions t4 and t5 model the possibility that this activity is a block activity or a subprocess activity: Transition t4 passes control to this block or subprocess, whereas transition t5 receives control back. Transition t6 models the possibility that the exit condition of the block activity evaluates to false. This transition needs to be removed if the activity is not a block activity. Transition t1 models the completion of the activity. Later on, we see that this transition starts all outgoing connectors, and that every outgoing control connectors is true if and only if its condition evaluates to true, and that every default connector evaluates to true if and only if every outgoing control connector evaluates to false.
Mappings

188

Figure 7.4 shows the net fragment for a connector. If the join condition of the preceding activity has evaluated to false, then its transition \( t_2 \) will have skipped the activity, which also skips this connector by immediately putting a token in the place \( p_2 \) of its succeeding activity, indicating that some connector of the succeeding activity evaluated to false.

If the join condition of the preceding activity has evaluated to true, then its transition \( t_1 \) will complete the activity, which puts a token into place \( p_1 \). For control connectors, transition \( t_1 \) models the situation that the connector condition evaluates to true, whereas transition \( t_2 \) models the situation that this condition evaluates to false. Thus, transition \( t_1 \) puts a token into the place \( p_1 \) of the succeeding activity, whereas transition \( t_2 \) puts a token into the place \( p_2 \) of this succeeding activity. Furthermore, an optional sibling default connector is notified of the evaluation result: If the condition has evaluated to true, then a token is put into the place \( p_2 \) of the default connector, otherwise, a token is put into the place \( p_3 \) of the default connector. The default connector evaluates to true if the condition of every sibling control connector has evaluated to false, that is, if the place \( p_3 \) of the default connector contains \( n \) tokens, where \( n \) equals the number of sibling control connectors. If the condition of every sibling control connectors has evaluated to true, that is, if the place \( p_2 \) of the default connector contains \( n \) tokens, then the default connector evaluates to false. If the condition of some sibling control connector has evaluated to true while the condition of some other sibling control connector has evaluated to false, then the default connector evaluates false. Thus, its transition \( t_2 \) should fire. This is taken care of by transition \( t_3 \) of the default connector, which behaves similar to the transitions \( t_7 \) and \( t_8 \) of an activity.

If no condition is associated to a control connector, then the connector has to be true. Thus, its transition \( t_2 \) needs to be removed. As men-

FIGURE 7.4. Net fragment for a connector.
tioned above, if the join condition of the preceding activity always evaluates to true, then the transition t2 of this activity needs to be removed. Thus, the arc from this transition to the place p2 of the succeeding activity needs to be removed as well.

Figures 7.5 and 7.6 show an example with a default connector and two sibling control connectors. For sake of simplicity, we assumed that the join condition of activity A always evaluates to true.
(Sub)process. Figure 7.7 shows the net fragment that is associated to a (sub)process with \( n \) start activities and \( m \) end activities. Note that because a start activity is not preceded by a (false) connector, the join condition of the start activity always evaluates to true. As a result, the net associated with a start activity does not contain a place \( p_2 \). Therefore, these places have already been omitted from Figure 7.7.

7.1.4 WF-nets

It is straightforward to check that a connected MQSeries workflow process definition is mapped onto a WF-net. In the case of the Original MQSeries process definition of Figure 7.1, the resulting WF-net contains 69 places and 65 transitions; in the case of the Revised MQSeries process definition of Figure 7.2, the resulting WF-net contains 102 places and 96 transitions. After applying the reduction rules as described in Section 4.4.1, the WF-nets contain 23 places and 19 transitions, and 33 places and 27 transitions.

7.1.5 Soundness

A WF-net that results from the mapping as described above, satisfies the option-to-complete and the proper-completion requirements of soundness, which is straightforward to show. Furthermore, it is also straightforward to show that the no-dead-tasks requirement also holds.
if default connectors are absent. Thus, only a default connector can cause a dead transition. Figure 7.8 shows two such examples.

In the leftmost MQSeries workflow process definition, the default connector is always false, because its sibling control connector is always true. As a result, among others, the transition $t_5$ corresponding to activity $B$ is dead.

In the rightmost MQSeries workflow process definition, either the control connector with condition $x > 0$ is true or the default connector, but not both. As a result, either activity $B$ or activity $C$ is false and gets skipped. However, activity $D$ needs both activity $B$ and $C$ to be true. As a result, among others, the transition $t_5$ corresponding to the activity $D$ is dead.

MQSeries Workflow is able to detect if a default connector has an unconditional sibling control connector. Thus, it can detect the error in the leftmost situation. However, MQSeries cannot detect an error in the (more complex) rightmost situation.

What results is a serious simplification of the diagnosis process that was introduced in Chapter 4, see Figure 4.14 on page 97. First of all, the resulting WF-net is always bounded, thus we can skip steps 3 to 9. Furthermore, a transition in this WF-net is either dead or live, thus, we can skip steps 11 to 13 too. As a result, we only need steps 2 (Workflow process definition?) and 10 (No dead tasks?). In case dead transitions are found, the list of dead transitions enables the designer to find and correct an anomalous default connector or a related error.

The WF-nets that result from mapping the sample MQSeries process definitions are both sound. However, if we remove all transition conditions from both processes, the disconnect activity will be dead as will the reconnect activity. As a result, the process would never come to an end. This can easily be detected and corrected because the transitions $t_1$ corresponding to both default connectors are dead.
7.1.6 Life-cycle inheritance

The mapped Revised MQSeries process definition is a subclass under life-cycle inheritance of the mapped Original MQSeries process definition: Hiding the three extending labels (archive, confirm, and reconnect) yields two branching bisimilar systems. Thus, the Revised MQSeries process definition is a proper extension of the Original MQSeries process definition.

We measured the computation times for both the backtracking algorithm and the exhaustive search algorithm, which were described in Chapter 5, using a 2.4 GHz computer with 256 Mb RAM running Windows 2000. For sake of completeness, we mention that the state space of the mapped Original MQSeries process definition contains 743 states, whereas the state space of the mapped Revised MQSeries process definition contains 2318 states. (Note that a lot of these states are transient states.) The backtracking algorithm needs to check branching bisimilarity only once, which takes approximately $5.81 \times 10^1$ seconds, with a 99% confidence interval of $1.09 \times 10^{-1}$ seconds. The exhaustive search algorithm needs to check branching bisimilarity eight times, which takes approximately $4.09 \times 10^2$ seconds with a 99% confidence interval of $2.09 \times 10^0$ seconds. Thus, in this case, the backtracking algorithm outperforms the exhaustive search algorithm by almost a factor 7 (where 8 would be optimal).

7.1.7 Summary

MQSeries process definitions can be successfully mapped onto nets, where connected MQSeries process definitions are mapped onto WF-nets. A mapped MQSeries process definition always satisfies the option-to-complete and proper-completion requirement of soundness, and can only violate the no-dead-transitions requirement if it contains default connectors. As a result, the diagnosis process presented in Chapter 4 can be simplified to two checks: is the mapped MQSeries process definition a WF-net and are there no dead transitions? If the net is not a WF-net, the designer should check connectedness of the MQSeries process definition, if dead transitions exist, the designer should check the defaults connectors.

If checking life-cycle inheritance for the given example, the backtracking algorithm needs to check branching bisimilarity only once, whereas the exhaustive search algorithm needs to check it eight times. Of course, this is mainly due to the fact that all extending labels need to be hidden whereas both algorithms prefer to block these labels. Nevertheless, it shows that the backtracking algorithm can ease the problem of checking life-cycle inheritance if an exhaustive search algorithm is used.
7.2 Staffware 2000

The following description of Staffware is based on [Sta97, Sta99, Sta02].

7.2.1 Processes

A Staffware process definition contains objects, links, and subprocesses. An object either represents a task, an entire subprocess, or a routing construct. A link connects two objects and represents the flow of control between these objects. Links that are connected to the right edge or the bottom edge of an object are outgoing links, while links that are connected to the left edge or the top edge are incoming links. As a result, we can partition the outgoing links into right links and bottom links, and the incoming links into left links and top links. With some exceptions, there is a difference between left links and top links, and between right links and bottom links. A process is started by a special point-of-entry, and is terminated if there is nothing left to do. Thus, termination of a process is implicit. While, in practice, some subtle differences might exist, we consider subprocesses to be processes.

First, we give an exhaustive list of objects that are relevant for the verification of a Staffware process definition.

**Start.** A start object represents the point-of-entry of the process definition and is represented visually by a traffic light. Every process definition has exactly one start object, every start object has exactly one right link, and that link should be a right link for either a step object, a subprocess object, or a complex router object. When a case is created, this link will be traversed.

**Step.** A step object represents a task in the process definition and is represented visually by a dog-eared form. The task is scheduled, that is, has an associated work item, if any one of its left links are traversed, unless the task was already scheduled. Thus, a task can only be scheduled once. As a result of this unless-clause, a step object does not necessarily behave like an XOR-join. We come back to this later on. If an activated work item is released, that is, if it is completed successfully, then the task is withdrawn, that is, unscheduled, and every right link is traversed. A task can also be withdrawn explicitly if any of the top links of the step object is traversed. Furthermore, a step object can have an associated deadline, in which case visually an alarm is added to the dog-eared form. If such a deadline expires, then every bottom link is traversed, and the task is withdrawn if and only if this is stated explicitly for this step object. Thus, we have two types of tasks with deadlines: The first type is withdrawn if the deadline expires, while the second type remains scheduled if the deadline expires.
Subprocess. A subprocess object represents an entire subprocess and is represented visually by three linked dog-eared forms and a yellow arrow. Although a subprocess is composite, its behavior resembles the behavior of a step object. Thus, it can be withdrawn, in which case every task and subprocess in the subprocess is withdrawn. Furthermore, it can have an associated deadline.

Complex router. A complex router represents a routing construct and is represented visually by a circle on an arrow. Except for the fact that a complex router does not represent a task, it behaves like a step object with only left links and right links. Thus, it cannot be withdrawn (no top links) and it has no associated deadline (no bottom links). As a result, a complex router behaves like a combination of an XOR-join and an AND-split.

Condition. A condition object represents a binary decision and is represented visually by a question mark on a diamond-shaped shield. A condition object has exactly one incoming link, and it is insensitive whether this is a left link or a top link. If the incoming link is traversed, the associated condition is evaluated. If this condition evaluates true, then all right links are traversed, otherwise, all bottom links are traversed.

Wait. A wait object represents a point-of-synchronization and is represented graphically by a combination of an hour-glass, an arrow, and an open door. A wait object has exactly one left link but can have up to sixteen top links. Every incoming link needs to be an outgoing link of either a step object, a subprocess object, or a complex router object. If its left link has been traversed and if the state of every step object, subprocess object, or complex router object corresponding to a top link its state is “released”, then all outgoing links are traversed. To understand object states, we first need to introduce four possible object actions: initialize, schedule, release, and withdraw. These action set the object state to “not processed”, “outstanding”, “released”, and “withdrawn” [Sta02]. Thus, the state of an object is “released” if and only if the last action performed on this object was a release action. As a result of this schema using object states, the wait object does not necessarily behave as an AND-join. We come back to this later on.

Router. Basically, a router object is a complex router object, except for the facts that a router object can have a top link and a bottom link and that it has exactly one incoming link and one outgoing link. For a router object, a top link is considered to be a left link, and a bottom link is considered to be a right link. Because it has only one incoming link and one outgoing link, a router object has no real use as a routing construct. In practice, it is used mainly to have some control over the link
routes, that is, by adding router objects on links, we can influence the routing of the link by moving the router object around.

**Stop.** A stop object represents a point-of-exit for a control path and is represented graphically by a stop sign. Note that the control flow can consist of many control paths. As a result, a process can contain many stop objects. However, every stop object is optional, and removing a stop object does not change the control flow. Thus, the only use of the stop object is to visualize a point-of-exit explicitly.

### 7.2.2 Sample processes

Figure 7.9 shows a Staffware (version 9.0) rendition of the Original process (see Section 3.2). After a case has been started, the client can connect to the portal. As a result of the Connect step, the variable orders is set to zero. Next, the client enters a loop in which he can order as many items as he likes. The Merge complex router symbolizes the entry point of every iteration of this loop. In every iteration, the client can either order an item or disconnect. If the client orders an item, the disconnect is withdrawn and vice versa. After the client disconnected, the case is terminated, which is symbolized by the optional stop sign. If the client ordered an item, the orders variable is incremented. After ordering an item, a check is made on this variable. If this variable equals one, that is, if this was the first order of a new shipment, then we schedule the Ship step and start the next iteration of the loop. If the order was an additional order, only a new iteration of the loop is started. When the company ships the pending orders, the orders variable is reset to zero.

![Diagram](https://via.placeholder.com/150)

**FIGURE 7.9.** The Original Staffware process definition.
Figure 7.10 shows a Staffware rendition of the Revised process (see Section 3.2). This process definition differs on some points from the Original Staffware process definition described above. First, a Confirm step has to precede the Ship step. Second, after disconnecting, either the client can reconnect or the company can archive. However, before archiving, the company should first complete any pending shipment. Therefore, if a pending shipment exists, we have to wait for it. If the company archives, the client cannot reconnect anymore; if the client reconnects, the company has to wait until the client disconnects again before archiving.

Please note that the choices in the Revised process definition are dependent, as both use the data element orders. Unfortunately, we were unable to model the Revised process in Staffware without using these dependent choices. As a result, abstraction from data leads to additional (and possibly unwanted) behavior. We come back to this later on, when we discuss the soundness of Staffware process definitions.

7.2.3 Mapping

The mapping presented here is based on the mapping presented in [AH00]. However, this mapping assumes that a step object behaves like an XOR-join and that a wait object behaves like an AND-join,
although this is not necessarily the case, as we observed earlier on. In Section 7.2.6, we comment on this decision and argue why it is acceptable to use this mapping.

First, we present the net fragment covering every object, second, we present the net fragment for the possible links between those objects, third, we present the net fragment for processes.

For sake of readability, we partition the set of objects into *top* objects and *non-top* objects. An object is a top object if it treats links from the top different from links to the left. Thus, step objects, subprocess objects, and wait objects are top objects. Similarly, we partition the set of object into *bottom* objects and *non-bottom* objects, where step object, subprocess objects, and condition objects are bottom objects.

Figure 7.11 shows the net fragment for every object except a subprocess object, while Figure 7.12 shows the net fragment for a subprocess object. Both figures contain place $p_1$ and transitions $t_1$ and $t_2$, which we explain first. Second, we explain place $p_2$ of Figure 7.11. Third and last, we explain place $p_3$ and transitions $t_3$, $t'$, and $t''$ of Figure 7.12.
A token in place $p_1$ indicates that the object is scheduled. Transition $t_1$ models the completion of the object, whereas transition $t_2$ models the expiration of the optional associated deadline. Transition $t_2$ needs to be removed if no deadline is associated to the object. As mentioned earlier, a deadline can either withdraw the object or leave the object scheduled. If the deadline withdraws the object, the arc from transition $t_2$ to place $p_1$ needs to be removed, otherwise it should not be removed. Transition $t_1$ puts tokens into the place of every object that corresponds to an outgoing right link, whereas transition $t_2$ puts tokens into the place of every object that corresponds to an outgoing bottom link.

In case of a wait object (see Figure 7.11), transition $t_1$ can only fire if every object corresponding to an incoming top link is released. As mentioned earlier, although a wait object does not necessarily correspond to an AND-join, it is acceptable to treat a wait object as an AND-join. As a result, we assume that transition $t_1$ can only fire after every transition corresponding to an incoming link has been completed, and, thus, that the incoming link has been traversed. Place $p_2$ of a wait object contains a token for every top incoming link that has been traversed. If all have been traversed and place $p_2$ contains $n$ tokens ($n$ equals the number of top incoming links), then transition $t_1$ is allowed to fire.

In case of a subprocess object (see Figure 7.12), an entire subprocess needs to be started and completed before the subprocess object can complete. Transition $t_3$ passes control to the subprocess and puts $x$ tokens into place $p_3$. Analogous to place $p_3$ corresponding to an XRL while_do routing element (see Figure 6.20 on page 165), this restricts the number of available controls for the subprocess to $x + 1$. Thus, when enacting the Staffware process definition, $x$ can be set to a large value, but when verifying it, $x$ should have a modest value. Note that this conforms to the mapping as presented in [AH00]. Every transition $t'$ in the subprocess that creates $b$ controls (because it produces $a + b$ tokens but consumes only $a$ tokens), needs to remove these controls from this place $p_3$. Every transition $t''$ in the subprocess that removes $b$ controls (because it consumes $a + b$ tokens but produces only $a$ tokens), needs to return these controls to place $p_3$. The object can only be completed if all controls have been returned, that is, if place $p_3$ contains $x + 1$ tokens.

Figure 7.13 shows the net fragments for a link from an object $A$ to an object $B$. Basically, the link can either be a withdraw link (object $A$ withdraws object $B$), a normal link (object $B$ needs to wait for object $A$), or a wait link (object $B$ additionally needs to wait for object $A$).
In case of a right withdraw link, if object A completes, step or subprocess object B needs to be withdrawn. Therefore, transition t1 of object A needs to remove the token from place p1 of object B.

In case of a bottom withdraw link, if the deadline for object A expires, step or subprocess object B needs to be withdrawn. Therefore, transition t2 of object A needs to remove the token from place p1 of object B.

In case of a right normal link, object B needs to wait until object A has been completed. Therefore, transition t1 of object A puts a token into place p1 of object B.

In case of a bottom normal link, object B needs to wait until the deadline of object A has expired. Therefore, transition t2 of object A puts a token into place p1 of object B.

In case of a right wait link, wait object B additionally needs to wait until object A has been completed. Therefore, transition t1 of object A puts a token into place p2 of object B.

In case of a bottom wait link, wait object B additionally needs to wait until the deadline of object A has expired. Therefore, transition t2 of object A puts a token into place p2 of object B.

Figure 7.14 shows the template for a process. As could be expected, this template resembles the template for subprocess a lot. The only differences are the existence of place p2, which signals completion of the entire process, and the absence of transition t2 (only subprocesses can have an associated deadline).

7.2.4 WF-net

By using the net fragments as described in the previous subsection, we obtain a net. However, in some cases the resulting net is not a WF-net.
We show that these cases are anomalies and should be reported as such.

Observe that the places $p_1$ and $p_2$ of the main process (see Figure 7.14) should be the only source place and sink place. Therefore, no other place should be a source place or a sink place.

For a non-subprocess object (see Figure 7.11), place $p_1$ can be a source place if there are no incoming normal links for the object. Because transition $t_1$ is always present, place $p_1$ cannot be a sink place. For a wait object, place $p_2$ can be a source place and a sink place if the wait object has no incoming wait links.

For a subprocess object (see Figure 7.12), place $p_1$ can be a source place if there are no incoming normal links for the object. Because transition $t_3$ is always present, place $p_1$ cannot be a sink place, place $p_3$ cannot be a source place, and place $p_1$ of the subprocess cannot be a source place. Because transition $t_1$ is always present, place $p_3$ cannot be a sink place and place $p_2$ of the subprocess cannot be a sink place.

Thus, for any object, place $p_1$ is a source place if there are no incoming normal links, and for a wait object, place $p_2$ is a source place and a sink place if there are no incoming wait links. Both these situations clearly correspond to anomalies in the Staffware process definition and should be reported. For the remainder of this subsection, we assume that such anomalies do not exist, that is, that we have exactly one source place and one sink place.

Unfortunately, having exactly one source place and sink place does not automatically result in a WF-net. However, it is straightforward to check that only a Staffware process that corresponds to a WF-net can be sound:

![Net fragment for a process.](image)

Suppose that in the resulting WF-net no path exists from the source place to some transition. Then no input place of that transition can get marked, thus, the transition either is dead or has no input places. However, according to the net fragments (see Figures 7.11 to 7.14), every transition has at least an input place, thus the transition is dead.

Suppose that in the resulting WF-net no path exists from some transition to the sink place. Then every token produced by this transition cannot be removed by any transition that leads to the sink place. However, according to the net fragments (see Figures 7.11 to 7.14), every transition has at least an output place, thus such a transition leads to improper completion.

As a result, we conclude that only an anomalous Staffware process does not correspond to a WF-net.

Figure 7.15 shows the WF-net that corresponds to the Revised Staffware process definition as shown in Figure 7.10. For sake of readabil-

\[ t_1^a \text{ arcs from place } p_3 \quad t_1^a \text{ arcs to place } p_3 \]

\textbf{FIGURE 7.15. The Revised Staffware process definition mapped to a WF-net.}
ity, we omitted the various arcs from and to the place p3. Instead, we used numbered superscripts and subscripts for the transitions that have arcs from or to this place, as indicated in the legend of the figure.

7.2.5 Soundness

It is straightforward to show that, because of the place p3 for every (sub)process, any mapped Staffware process definition satisfies the proper-completion requirement: Similar to a mapped XRL process definitions (see Section 6.4), a mapped Staffware process definition can be covered by a positive place invariant. However, the other two soundness requirements can be violated. For example, the mapped Revised Staffware process definition (see Figure 7.15) does not have the option to complete after having fired the transition sequence (t1, Connect/t1, Merge/t1, Disconnect/t1, orders=0/t2), which leads to marking \([p3^x, \text{Dummy/p1}]\). From this marking on, the transition wait/t1 will always block.

Note that the above-mentioned error is the direct result of abstracting from data. After all, after having fired the transition sequence (t1, Connect/t1, Merge/t1, Disconnect/t1), no orders have been placed, hence, the condition orders = 0 should evaluate to true, thus, transition orders=0/t1 should fire instead of transition orders=0/t2. However, as argued in Chapter 2 (Section 2.2), we consider data to be volatile. As a result, after having fired the transition sequence (t1, Connect/t1, Merge/t1, Disconnect/t1) we have no guarantee that condition orders = 0 evaluates to true, and, thus, that the transition sequence (t1, Connect/t1, Merge/t1, Disconnect/t1, orders=0/t2) can also occur in the Revised Staffware process definition.

Basically, a mapped Staffware process can violate the option-to-complete requirement for two reasons:

1. a wait blocks indefinitely, or
2. an additional control is needed, but the corresponding place p3 is empty.

In the first case, the problem is clearly related to the option-to-complete requirement. However, in the second case, the problem is more related to the proper-completion requirement, because, obviously, we have a loop that can keep on spawning controls.

Obviously, when some wait object blocks indefinitely, the following objects that depend on this wait object will be dead. Because we abstracted from data, only a blocking wait object can cause dead objects, and, hence, dead transitions.
Like in the case of MQSeries Workflow, the diagnosis process as presented in Chapter 4 can be simplified, although to a lesser extent: Steps 11 to 13 cannot be removed because a resulting WF-net might not have the option to complete. If dead transitions are detected, then the designer can be guided towards correcting the error because the transition t1 of some wait object will be dead. Obviously, one of such wait objects must be the cause of the error. If some transitions are live while none is dead, then the designer can also be guided towards correcting the error because either:

- there exists a marking in which the place p3 corresponding to some (sub)process is empty, which indicates that this process can keep on spawning new control paths, or
- a transition t1 corresponding to some wait object is non-live, which indicates that this wait object might block in some situations.

7.2.6 Semantical issues

Some Staffware process designers might argue that the mapping presented in this section should not be used because it is incorrect, as the resulting WF-net might behave differently from the Staffware process on the enactment server. We agree that the resulting WF-net can behave differently. However, in our point of view, the differences in behavior that may arise between a Staffware process and its mapped WF-net are the result of anomalies in the Staffware process semantics. This section explains this point of view.

First of all, the wait object does not always behave like an AND-join, that is, it does not always synchronize on top links. In fact [Sta02], the wait object waits until all objects connected to its top have the status “released”. Recall that the statuses of an object can be “not processed”, “outstanding”, “released”, or “withdrawn”. If a wait object occurs inside a loop, then some object that is connected to its top can still have the status “released”, because it has not been scheduled yet in the new iteration. Figure 7.16 shows an example for this. After task A has been released for the second time, both tasks B and C will be scheduled and, thus, have the status “outstanding”, but tasks E and S will not be scheduled (yet) and, thus, still have the status “released”. Thus, if task C is released before task B, then the wait object will not wait for tasks E or S, because they still has the status “released”. We consider this to be an anomaly, because the wait object would have waited on task S if task B would have been absent. Thus, task B effects the wait object, although it is not (directly) connected to it. To repair this anomaly, Staffware provides a function, called “SETSTEPSTATUS”, that allows us to set the status of any object to either “outstanding” or “released” [Sta02]. In our opinion, this solution is even worse than the problem, as it opens the door to incomprehensible object interrelationships. Because we
map a wait object onto an AND-join, the Staffware process definition shown in Figure 7.16 is mapped onto an unsound WF-net, as the corresponding synchronizing transition can be executed only once. Thus, this anomaly will be detected.

The same status concept seems to be related to a second anomaly: a task can correspond to at most one work item, that is, it can only have the status “outstanding” once. Thus, if a step object already has a corresponding work item and if a left link is traversed, then nothing will happen. As a result, the order in which some step objects are released determines how many times some other step object needs to be executed. Figure 7.17 shows an example for this. In this example, task S can be executed once, twice, or thrice. Typical release sequences for these possibilities are:

1. \((A, B, C, D, S)\),
2. \((A, B, S, C, D, S)\), and
3. \((A, B, S, C, S, D, S)\).

FIGURE 7.16. An anomalous example for the wait object.

FIGURE 7.17. An anomalous example for the join.
Note that because an object corresponds to at most one work item, the number of active control paths is limited to the number of objects. Thus, the number of objects in a (sub)process is a sufficient value for the number of tokens to be inserted in its corresponding place $p_3$, that it, it is an upper bound for the variable $x$ in Figures 7.12 and 7.14. Because we map a step object onto an XOR-join, the Staffware process definition shown in Figure 7.17 is mapped onto an unsound WF-net, this anomaly can be detected, as the place corresponding to task $S$ is unbounded (in the short-circuited WF-net).

As a result of both anomalies, the wait object does not behave like an AND-join, and, among others, the step object does not correspond to an XOR-join, as was assumed in the mapping. Thus, the resulting WF-net will behave differently from the original Staffware process. However, as indicated, we consider this to be caused by anomalies in Staffware, which need to be detected. In the case of Figure 7.16, the mapped wait object will block eventually, and thus cause a violation of the option-to-complete requirement. In the case of Figure 7.17, the mapped step object will contain a place that is not safe. Thus, both anomalies can be detected by verifying the resulting WF-net.

### 7.2.7 Summary

This section has presented a mapping from Staffware process definitions to nets. However, a resulting net does not necessarily behave identical to its Staffware process definition. This is caused by some peculiarities of some Staffware constructs that we regard to be anomalies.

Not every Staffware process definition is mapped onto a WF-net. However, only incorrect Staffware process definitions are not mapped onto WF-nets. Such incorrect Staffware process definitions typically include:

- unconnected parts,
- wait objects without incoming top links, and
- objects without incoming links.

Although a mapped WF-net might behave different from its Staffware process definition, we believe that we can successfully use the presented mapping to verify soundness of Staffware process definitions that are mapped onto WF-nets. For Staffware process definitions, the diagnosis process as presented in Chapter 4 can be simplified, because every mapped Staffware process definition is bounded.

Our Revised Staffware process definition (see Figure 7.10) is mapped onto an unsound WF-net, which is mainly due to the fact that we had to
use data to model the Revised process in Staffware. As a result, we could not check whether the mapped Revised Staffware process definition is a subclass under life-cycle inheritance of the mapped Original Staffware process definition (see Figure 7.9).

7.3 COSA Workflow

The following description of COSA Workflow is based on [SL98, Baa99].

7.3.1 Processes

The control flow of a COSA Workflow [Baa99] process definition contains processes, activities, conditions, and transitions. For sake of clarity, and to avoid ambiguity later on, henceforth, we refer to conditions as requirements, and to transitions as transfers. A COSA process definition resembles a Petri net: activities correspond to transitions, requirements correspond to places, and transfers correspond to arcs. (Please note that a transition in COSA does not correspond to a transition in a Petri-net, but to an arc. For this reason, we refer to transitions in COSA as transfers.) As a result, it is relatively straightforward to map COSA process definitions onto nets by mapping:

- every activity onto a transition,
- every requirement onto a place, and
- every transfer onto an arc.

However, there are some differences between COSA process definitions and our nets:

- Instead of an input (output) place, a COSA process definition has a start (end) activity.
- an activity can model an entire (sub)process, by referring to a COSA process definition model.
- a transfer allows for a condition, which might allow activities to start even if some input requirements are not fulfilled. (Because transfers can have conditions, we refer to condition objects in COSA as requirements.)

The third difference, transfer conditions, needs some additional explanation, because COSA allows for the special expression “&CONDITION&” in those transfer conditions. For an input transfer (a transfer that connects a requirement to an activity), this expression evaluates to true if and only if the requirement is fulfilled, that is, if it contains tokens; for an output transfer (a transfer that connects an activity to a
requirement), this expression evaluates to true if and only if the activity has been completed. By default, every transfer has the condition “&CONDITION&”. As a result, an activity can only start if and only if all its input requirements are fulfilled (contain tokens), and an activity fulfills (puts tokens into) its output requirements if and only if it has been completed. The expression “&CONDITION&” can also be used to specify advanced behavior. For example, if an input transfer has the condition1 “&CONDITION& && \( P \)”, for some predicate \( P \), then the corresponding activity can only start if the corresponding requirement is fulfilled and predicate \( P \) holds. However, if an input transfer would have the condition “\( P \)”, then the activity could start regardless whether the requirement is fulfilled. If an output transfer has a condition “\( P \)”, then the activity fulfills the requirement if and only if predicate \( P \) holds. However, if an output transfer has a condition “&CONDITION& || \( P \)”, then the activity can fulfill the requirement without having been started or completed.

7.3.2 Sample processes

Figure 7.18 shows a COSA rendition of the Original process definition (see Figure 3.4 on page 46), whereas Figure 7.19 shows this for the Revised process definition (see Figure 3.3 on page 44). Because COSA requires exactly one output transition, in the Original COSA process definition we have added a transition named wrap_up. Furthermore,

1. COSA Workflow uses a C-like syntax for its transition conditions: The operator “&&” corresponds to a logical AND and the operator “||” corresponds to a logical OR.
COSA does not allow two activities to have identical names. Therefore, we have augmented certain names.

7.3.3 Mapping

Most of the mapping is straightforward, as COSA process definitions resemble Petri nets. Therefore, this section describes the differences between COSA process definitions and our nets as mentioned above. Figure 7.20 shows that it is straightforward to add a source place (p1) and a sink place (p2) to a COSA process definition. Figure 7.21 shows that we can replace a transition that models an entire net by two transitions (t1 and t2) and one place (p1). Transition passes control to the subnet, whereas transition t2 receives control from that subnet. Place p1 ensures that only transitions that have passed control to a subnet can receive control from that subnet.

FIGURE 7.19. The Revised COSA Workflow process definition.

FIGURE 7.20. Adding a source and sink place to a COSA process definition.
In case an input transfer has no condition or a condition of the form “&CONDITION& && P”, for some predicate P, then the corresponding activity needs the corresponding requirement to be fulfilled. However, if the expression “&CONDITION&” is absent from the condition, then the activity could start even if the requirement is not fulfilled. Take, for example, the activity shown in Figure 7.22. This activity can start if requirements input2 and input3 are fulfilled, but whether input1 should be fulfilled depends on data. Because we abstract from data, we cannot tell whether or not input1 should be fulfilled.

In case an output transfer has no condition or a condition of the form “P”, for some predicate P, then the corresponding activity needs to fulfill the corresponding requirement if it completes. However, if the expression is of the form “&CONDITION& || P”, then the activity might fulfill the requirement if it is not completing. Take, for example, the activity shown in Figure 7.23. If this activity completes, then it needs to fulfill requirements output2 and output3, and possibly output1 as well. Furthermore, after we have abstracted from data, requirement output2 can be fulfilled at any time.

Figure 7.23 shows how we map the activity of Figure 7.23 to a net fragment. Note that we introduce a dedicated transition t5 that can put
a token into place $p_5$ (which corresponds to the requirement output2) as long as the process is running.

In certain cases, the designer of the COSA process definition might be able to tell that a certain activity behaves like an AND-join, an OR-join, or an XOR-join. This information might be of value for the verification of the COSA process definition. For this reason, we have extended COSA with three analysis keywords: and_join, or_join, and xor_join. Figure 7.24 shows the Activity Script dialog for an activity. In this dialog, we first need to select the radio button named Analysis, followed by entering the specific keyword in the text box. In this case, we have specified that this activity acts as an OR-join. The net fragments for an AND-join or an XOR-join are straightforward, and not shown here. However, the net fragment for an OR-join is more complicated, as for an OR-join we have to take the conditions into account.

Similar to the keywords and_join, or_join, and xor_join, keywords and_split, or_split, and xor_split exist to specify in the Activity Script dialog that an activity behaves like an AND-join, OR-join, or XOR-join. Again, the net fragments for the AND-split and the XOR-split are straightforward and not shown here, while the net fragment for the OR-split needs to take the conditions into account.

### 7.3.4 WF-net

A COSA process definition needs to satisfy the following structural requirements [Baa99]:

![Diagram](image-url)
A COSA process definition is connected.
A COSA process definition contains exactly one start activity, which has output transfers but no input transfers.
A COSA process definition should contain exactly one end activity, which has input transfers but no output transfers.
Every requirement has input and output transfers.
Every activity has output requirements.

However, an activity may have no input requirements. As a result, the mapping may result in a net that is not a WF-net. Furthermore, as we have seen in the previous section, the mapping may insert source transitions (see transition t5 in Figure 7.23). At this point, we could decide to change the mapping in such a way that every transition will have an input place. For example, we could introduce a place that acts as input and output place for every transition that would otherwise be a source transition, and we could put a token into this place when the instance starts (see transition t1 in Figure 7.20) and remove this token when the instance completes (see transition t2 in Figure 7.20). However, adding this place only results in an unsound WF-net, which only shifts the diagnosis to a later (and more complex) stage.

7.3.5 Soundness

It is straightforward to check, that soundness can only hold if the following requirements on conditions of transfers are met (because otherwise substates exist, see Section 4.5.8):

FIGURE 7.24. Specifying in COSA that an activity behaves like an OR-join.
If an activity is specified to be an OR-join, then the condition of every input transfer should be empty or should be of the form "&CONDITION& && P", for some predicate P. (Note that this results in an OR-join that behaves like an AND-join.)

If an activity is specified to be an OR-split, then the condition of every output transfer should be empty or should not be of the form "&CONDITION& || P", for some predicate P. (Note that this results in an OR-split that behaves like an AND-split.)

The example COSA process definitions are mapped onto WF-nets that are behaviorally identical (after hiding the activities named dummy and wrap_up) to the WF-nets shown in Figure 3.4 on page 46 and Figure 3.3 on page 44. For these WF-nets, soundness and life-cycle inheritance have already been discussed in detail. Therefore, we do not discuss soundness and life-cycle inheritance for the example COSA process definitions in depth here.

### 7.3.6 Semantical issues

As mentioned in the introduction of this chapter, COSA process definitions are safe workflow process definitions. Thus, a requirement can either be fulfilled or not, but it can not be fulfilled twice. In the previous section on Staffware, we argue that this can lead to anomalous behavior, as the number of times a certain activity has to be executed depends on the order in which activities are executed. For the same reason as in the Staffware case, that is, to be able to detect these anomalies, our mapping does not transfer this safe behavior.

### 7.3.7 Summary

COSA process definitions can be mapped onto nets, but not every COSA process definition is mapped onto a WF-net. However, a COSA process definition that is not mapped onto WF-net would be unsound if we would map it onto a WF-net.

The diagnosis process as presented in Chapter 4 cannot be simplified in the case of COSA process definitions, as every WF-net can be modeled using COSA.

### 7.4 Protos

The following description of the Protos BPR is based on [Pal97, Wav02].
The control flow of a Protos process definition contains activities, statuses, and connections. Like the COSA Workflow WFMS, Protos \cite{Wav02, Pal97} generates process definitions that resemble Petri nets: activities correspond to transitions, statuses correspond to places, and connections correspond to arcs. Figure 7.25 shows how the Original process can be modeled using Protos.

Like COSA Workflow, Protos:
- prescribes that there exists start and end activities,
- allows for connection conditions,
- allows for activities that refer to an entire process definition,
- allows for specifying whether an activity behaves like an AND-join, OR-join, or XOR-join, and whether it behaves like an AND-split, OR-split, or XOR-split. Figure 7.26 shows the Simulation tab for Activity Properties dialog for the activity named Order. However, this Simulation tab is only present if certain Protos add-ons are installed, like, for example, the Woflan add-on.

However, unlike COSA Workflow, Protos:
- allows for multiple start and end activities,
- does not allow for an activity to start without having all its input statuses fulfilled, and to fulfill an output status if it is not completing.

### 7.4.2 Mapping

Given the above-mentioned observations and the mapping from COSA Workflow onto nets, the mapping from Protos process definitions onto
Mappings

nets is similar to the mapping from COSA Workflow process definitions onto WF-nets. However, some differences are noteworthy:

- the introduced source place is connected to every start activity,
- every end activity is connected to the introduced sink place,
- conditions on input connectors cannot lead to an OR-join, and
- conditions on output connectors cannot lead to an activity that fulfills an output status if it did not complete.

Thus, the Protos mapping corresponds basically to a restricted COSA mapping. For this reason, we do not investigate the WF-net, soundness, and life-cycle inheritance properties for the Protos mapping, as the results would be analogous to the results from the COSA mapping. Furthermore, the same conclusions apply for the Protos mapping as for the COSA mapping.

FIGURE 7.26. Specifying join (In) and split (Out) behavior in Protos.
7.5 Conclusions

In this chapter, we have presented mappings from processes defined using the WFMSs MQSeries Workflow, Staffware, and COSA and the BPR tool Protos, onto WF-nets. We also have investigated whether the mapping always results in a WF-net, in which cases the resulting WF-net is unsound, and we have shown that the diagnosis process from Chapter 4 can be used to diagnose an unsound WF-net.

Furthermore, in the case of MQSeries Workflow, we have seen that
- only unconnected control flow models result in non-WF-nets,
- the resulting WF-net always has the option to complete and proper completion,
- only an anomalous default connector can result in an unsound WF-net, and
- the diagnosis process can be simplified to a large extent.

For Staffware process definitions, we have seen that
- only unconnected Staffware processes result in non-WF-nets,
- the resulting WF-net always has proper completion,
- a blocking wait or a loop that can keep spawning new control paths violates the option-to-complete requirement,
- the resulting WF-net might behave different from the original Staffware process, but that this is due to anomalous Staffware constructs which can be detected and diagnosed, and
- the diagnosis process can be simplified to some extent.

Finally, for COSA Workflow and Protos process definitions we have seen that
- only processes in which some object is not on a path from some start transition to some end transition result in non-WF-nets,
- the resulting WF-net can violate any soundness requirement, and, thus,
- we cannot simplify the diagnosis process.

Table 7.1 summarizes these results by displaying their effect on the diagnosis process as presented in Chapter 4 (see also Figure 4.14 on page 97). For sake of comparison, we also included XRL (see previous chapter) in this summary. According to this table, an MQSeries process definition:
- is a workflow process definition if it is connected,
is unlikely to be mapped onto WF-net that has a thread of control
cover (only trivial examples may have such a cover),
is likely to be mapped onto WF-net that contains confusions and
mismatches (only trivial examples will not contain them),
is unlikely to be mapped onto WF-net that has a uniform invariant
cover (only trivial examples may have such a cover),
is mapped onto a WF-net that has a weighted invariant cover,
may or may not contain dead tasks (which depends heavily on the
process definition at hand), and
is mapped onto a WF-net that contains no non-live tasks.

Note that the main contribution of the diagnosis process are the steps
labeled “Depends”. Thus, Table 7.1 clearly shows that the diagnosis
process for XRL and MQSeries Workflow can be simplified to a large
extent, for Staffware to some extent, and (almost) not at all for COSA
Workflow (which also includes Protos).

<table>
<thead>
<tr>
<th>Step number</th>
<th>XRL</th>
<th>MQ Series Workflow</th>
<th>Staffware</th>
<th>COSA Workflow</th>
</tr>
</thead>
<tbody>
<tr>
<td>2: workflow process</td>
<td>Yes</td>
<td>If connected</td>
<td>If connected</td>
<td>If strongly connected (after short-circuiting)</td>
</tr>
<tr>
<td>definition?</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3: thread of control</td>
<td>Yes</td>
<td>Unlikely</td>
<td>Unlikely</td>
<td>Depends</td>
</tr>
<tr>
<td>cover?</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4: confusions and mis-</td>
<td>n/a</td>
<td>Likely</td>
<td>If no withdraw links</td>
<td>Depends</td>
</tr>
<tr>
<td>matches?</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5: uniform invariant</td>
<td>n/a</td>
<td>Unlikely</td>
<td>Unlikely</td>
<td>Depends</td>
</tr>
<tr>
<td>cover?</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6: weighted invariant</td>
<td>n/a</td>
<td>Yes</td>
<td>Yes</td>
<td>Depends</td>
</tr>
<tr>
<td>cover</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7: no improper condi-</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>Depends</td>
</tr>
<tr>
<td>tions?</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8: no substates?</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>Depends</td>
</tr>
<tr>
<td>9: improper scenarios!</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>Depends</td>
</tr>
<tr>
<td>10: no dead tasks?</td>
<td>Depends</td>
<td>Depends</td>
<td>Depends</td>
<td>Depends</td>
</tr>
<tr>
<td>11: no non-live tasks?</td>
<td>Yes</td>
<td>Yes</td>
<td>Depends</td>
<td>Depends</td>
</tr>
<tr>
<td>12: non-live tasks!</td>
<td>n/a</td>
<td>n/a</td>
<td>Depends</td>
<td>Depends</td>
</tr>
<tr>
<td>13: locking scenarios!</td>
<td>n/a</td>
<td>n/a</td>
<td>Depends</td>
<td>Depends</td>
</tr>
</tbody>
</table>

TABLE 7.1. The effect of the presented mappings on the diagnosis process.
This chapter describes the tool Woflan. Woflan is a tool that can verify workflow process definitions. The need for such a tool was signaled in Chapter 2. The development of the tool Woflan started at the end of 1996 and the first version was released in 1997 [AHV97]. Basically, Woflan takes a workflow process definition from some workflow management system, converts it into a net, and checks whether or not the net is a sound WF-net or whether one sound WF-net is a subclass of a second sound WF-net under life-cycle inheritance. Furthermore, using standard net-analysis techniques as well as those tailored to WF-nets, the tool provides diagnostic information in case the net is not a sound WF-net.

The remainder of this chapter is organized as follows. First, we discuss the architecture of the latest released version of Woflan (release 2.2). Second, we discuss the look and feel of the Windows application that is part of this release. Third, we discuss a Web service that has been build using this release, but is not part of it. Fourth, we discuss ongoing work on Woflan, that is, we discuss new features for the next release of Woflan. Among these new features is the backtracking algorithm as presented in Chapter 5. Last, we conclude the chapter.

8.1 Architecture

Figure 8.1 shows the architecture of Woflan 2.2 as an UML component diagram. This diagram shows that, at the moment, Woflan 2.2 consists of one executable (wofapp.exe) and a number of libraries. This section discusses the components shown in Figure 8.1 in a bottom-up way.
All components have been implemented in C++. Although the initial version of Woflan was implemented on a UNIX workstation, the Woflan software is now being developed on a Windows platform using the Microsoft Visual C++ 6.0 suite. However, we feel confident that all static libraries compile without any problems on other platforms like, for example, UNIX. As a result, only the dynamic library and both executables need to be rewritten if Woflan needs to be transferred to a non-Windows platform.

### 8.1.1 Static library grcps.lib

This static library contains a branching bisimilarity algorithm that is based on [GV90]. This library exports five functions that only use basic types like abstract handles (void *), strings (char *), and numbers (int). Figure 8.2 shows the declarations of these functions. The peculiar name of the library originates from [GV90], where the algorithm solves basically the problem of Generalized Relational Coarsest Partitioning with Stuttering.
Using this library, we can construct a combined state space from multiple reachability graphs. On this combined state space, we can determine, using the above mentioned branching bisimilarity algorithm, which states are branching bisimilar. If all initial states are branching bisimilar, then all corresponding systems are branching bisimilar. The library woflib.lib uses this library to check whether or not two systems are branching bisimilar.

**GRCPS_state.** The function GRCPS_state adds a state (a node) to the combined state space, and returns an abstract handle to this state.

**GRCPS_transition.** The function GRCPS_transition expects abstract handles to two states in the combined state space and a transition label, and adds a transition (an edge) from the first state to the second state with the given label to the combined state space.

**GRCPS_init.** After the entire combined state space has been constructed by functions GRCPS_state and GRCPS_transition, the function GRCPS_init initializes the branching bisimilarity check (for example, it deals with inert cycles in the state space, as the algorithm presented in [GV90] cannot deal with such cycles). The function expects two labels: the first label is the label that corresponds to a hidden label, that is, \( \tau \), whereas the second label corresponds to the label that corresponds to a blocked label.

Note that we effectively hide a label by renaming it to the first label, and we effectively block a label by renaming it to the second label. As a result of blocking a label by renaming it instead of by removing all transitions associated to it, we only need to construct the state space once if checking life-cycle inheritance.

**GRCPS_mill.** After GRCPS_init has initialized the branching bisimilarity check, this function actually checks for branching bisimilarity.

```c
1 void *GRCPS_state(  
2 );  
3 void GRCPS_transition(  
4 void *from,  
5 void *to,  
6 const char *label  
7 );  
8 void GRCPS_init(  
9 const char *hidden,  
10 const char *blocked  
11 );  
12 void GRCPS_mill(  
13 );  
14 int GRCPS_isbb(  
15 void *one,  
16 void *two  
17 );
```

**FIGURE 8.2.** Declarations of functions exported by static library grcps.lib.
After function GRCPS_mill has checked for branching bisimilarity, this function returns whether two states are branching bisimilar. As input, this function expects the abstract handles to these two states.

8.1.2 Static library woflib.lib

This library is the core of Woflan and contains many Petri-net-based analysis routines. First of all, this library exports the class AnaNet (Analysis Net), which contains methods to check properties that are related to the soundness of a given (WF-)net. Second this library exports a number of class that are necessary to contain the input parameters and the results of these methods. Figure 8.3 shows, for example, the classes exported by this library to contain the results of the methods. Below follows a brief overview of these classes and why they are needed.

**P_Array.** The class P_Array is needed to contain, for example, the source places, the sink places, or unbounded places.
T_Array. The class T_Array is needed to contain, for example, source transitions, sink transitions, and dead transitions.

N_Array. The class N_Array is needed to contain, for example, nodes that are not connected to a source or sink place, or nodes that are not covered by any S-component.

W_Array. The class W_Array is needed to contain weighted arcs.

NL_Array. The class NL_Array is needed to contain, for example, non-free-choice clusters, or S-components.

NPL_Array. The class NPL_Array is needed to contain PT-handles or TP-handles.

PI_Array. The class PI_Array is needed to contain place invariants.

TI_Array. The class TI_Array is needed to contain transition invariants.

If needed, the woflib.lib library generates reachability graphs, coverability graphs, and/or minimal coverability graphs. Furthermore, this library exports functions to check, using the library grcps.lib as mentioned above, whether or not one sound WF-net is a subclass under any of the four inheritance relations (see Chapter 1) of another sound WF-net. Last, this library exports functions to load TPN format file. The TPN format originates from [Aal92] and is described in Appendix B.

Note that this library does not implement the diagnosis process as presented in Chapter 4, which is implemented in the Windows application itself (and not in any of the libraries).

8.1.3 Static library wofdll.lib

As mentioned above, the library woflib.lib exports a multitude of classes and methods. As a result, any component that uses this library needs to be aware of all these classes and methods. For this reason, we introduce a wrapper library that wraps the classes and methods exported by the woflib.lib library into a library that only exports three functions: WoflanOpen, WoflanClose, and WoflanInfo, and that only uses basic types like nonnegative numbers (unsigned) and strings (char *). Figure 8.4 shows the declarations for these functions.

After loading a net using WoflanOpen, one can check properties of the loaded net using any number of WoflanInfo calls. After all necessary properties have been checked, the loaded net can be unloaded using WoflanClose. As a result of this wrapping, the interface between the
executables and the underlying libraries is restricted to three functions, instead of a multitude of functions.

WoflanOpen. The function WoflanOpen expects the name of the file to open as its only argument. When called, it will scan that file for a net (TPN format), load that net, and return a handle to that net. If the file does not contain a net, most likely, an empty net will be loaded.

WoflanClose. The function WoflanClose expects a handle to a loaded net as its only argument. When called, it will unload that net, and return 0.

WoflanInfo. The function WoflanInfo expects up to four arguments, of which the first two are mandatory. The first argument should be a handle to some loaded net, while the second argument should be some number indicating the property that is requested about that net. If, for example, the name of some place is requested, then the third argument should be the index to that place. Likewise, if the name of some place in some place invariant is requested, then the third argument should be the index to that place invariant and the fourth argument should be the index in that place invariant to that place.

A full description of the function WoflanInfo and its arguments is beyond the scope of this chapter, but can be found in Appendix A.

8.1.4 Dynamic library wofdll.dll

The library wofdll.dll results if the library wofdll.lib is wrapped as Windows DLL. Any Windows application can use this DLL as a back office to check properties that are relevant for soundness and to check any of the four inheritance relations. As a result, any Windows-based WFMS could use this DLL to check soundness and inheritance on its proprietary process definitions, provided that they can map these process definitions onto nets. As mentioned earlier, the libraries grcps.lib, woflib.lib, and wofdll.lib are not restricted to Windows platforms.
This library exports a function that loads a COSA script file (*.scr), maps the loaded COSA process definition onto a net given the net fragments as presented in Chapter 7, and generates a TPN file (*.tpn) containing the resulting net. Figure 8.5 shows how this library can be used: An application (like wofapp.exe) that needs to have a COSA script file mapped onto a net, calls the co2tpn function with the name of the COSA script file as argument (not shown). As a result, a TPN file is generated, which is loaded by the application by calling the WoflanOpen function with the name of the TPN file. After the net has been loaded, the application can diagnose the net by calling the WoflanInfo function. Note that the fact that wofdll.dll uses wofdll.lib and possibly also grcps.lib is not shown in Figure 8.5, because this would only result in an unnecessary complicated figure.

In case the COSA process definition contains activities that refer to other COSA process definitions, the library tries to load the corresponding COSA script file. However, unfortunately, the name of a COSA process definition may differ from the name of its corresponding COSA script file. Although the COSA system is able to keep track of these differences, the co2tpn.lib library is not. As a result, it will fail to load a COSA process definition if the name of its corresponding COSA script file differs.
For the Protos BPR tool, a Woflan add-on exists that enables the user of Protos to verify a modeled Protos process definition using Woflan. This add-on exports the Protos process definition to a COSA script file. As a result, the co2tpn.lib library is also used to load Protos process definition and map it onto a net.

8.1.6 Static library xfr2tpn.lib

Similar to the co2tpn library, the xfr2tpn library exports a function that loads a Staffware procedure file (*.xfr), maps the loaded Staffware process definition onto a net, and generates a TPN file (*.tpn) containing the resulting net. This library was implemented based on Staffware 2000 (8.0) process definitions. As a result, because sub procedures are a new feature in version 9.0, this library does not support sub procedures yet. Nevertheless, it is straightforward to add this functionality, given the net fragments as presented in Chapter 7.

8.1.7 Static library ...2tpn.lib

This library acts a placeholder for the libraries that can load workflow process definition from other WFMSs like, for example, IBM MQSeries, map the workflow process definition to a WF-net, and generate a TPN file containing that WF-net.

8.1.8 Executable wofapp.exe

This executable is the Windows-based Woflan 2.2 application. This application implements the diagnosis process as presented in Chapter 4 and uses all above mentioned libraries to check relevant net properties. Section 8.2 discusses this application in detail.

8.2 Windows application

The terminology used by the Windows application (wofapp.exe) is based on the terminology used by the WfMC [WFM96]. However, to avoid confusion within the Petri-net community, we use the term condition instead of transition to describe places. Table 8.1 shows the relation between the workflow terminology [WFM96] used in this paper and Petri-net terminology. Note that many of the properties of a process definition are defined on the short-circuited WF-net, which avoids mentioning over and over again the fact that we have to short-circuit the WF-net.

The Windows application implements the diagnosis process as introduced in Chapter 4. To show how the Windows application operates,
8.2.1 Diagnosis process

First, we need to export the Revised process definition on the Staffware server. The resulting XFR file can be mapped onto a process definition (that is, a net) by Woflan’s xfr2tpn library. To do so, we open the file, see Figure 8.6, which results in the dialog shown in Figure 8.7. This dialog shows the content of the TPN file that results from mapping the Staffware Revised process definition. The next step is to check whether the process definition is a workflow process definition, which is shown by Figure 8.8. The process definition is obviously a workflow process definition, therefore, we continue with checking whether the short-circuited net is covered by threads of control, which is shown by
Figure 8.9. This figure also shows which places are not covered by any thread of control, which might be useful diagnostic information when one expects a thread-of-control cover. However, because of the conditions p3 (see Figure 7.12 on page 197 and Figure 7.14 on page 200), chances are very slim that a mapped Staffware process has a thread-of-control cover. Therefore, we did not expect such a cover, and proceed to check whether confusions and mismatches exist. If either of them does not exist, the workflow process definition cannot be sound, as was proved in Chapter 4. This step is shown in Figure 8.10. As we can see, both confusions and mismatches exist, therefore, the workflow process definition can still be sound. According to the diagnosis process, the next step would be to check whether all places can be covered by uniform invariants. However, like with the thread-of-control cover,
chances are very slim that such a cover exists, and given the fact that from Figure 8.7 we may conclude that the variable x (see Figure 7.14 on page 200) equals 7 and that, thus, the chance on such a cover is non-existent. Therefore, we skip Step 5 in this explanation, and move on to Step 6, which is shown in Figure 8.11. As expected, a weighted-invariant cover exists, and we may conclude that the workflow process definition is proper. This leaves us to check whether the workflow process definition contains dead tasks and whether it has the option to com-

FIGURE 8.8. Step 2: The process definition is a workflow process definition.

FIGURE 8.9. Step 3: There is no thread-of-control cover.
Woflan complete. First, we check whether any task is dead, which is shown in Figure 8.12. Apparently, no task is dead, thus, we move on to Step 11 to check whether every task is live. This step is shown in Figure 8.13. Obviously, there are quite a number of non-live transitions, among which the short-circuited transition (called EXTENSION). To summarize the results so far:

- completion is proper,

---

**FIGURE 8.10.** Step 4: There are confusions and mismatches.

**FIGURE 8.11.** Step 6: There is a weighted-invariant cover, hence completion is proper.
As a result, this workflow process definition is not sound, because the option-to-complete requirement is violated.

To diagnose why the option-to-complete requirement is violated, we proceed with Step 13, which gives us the relevant locking scenarios.
Figure 8.14 shows us that there are 54 locking scenarios, where the first scenario has been unfolded. From this first locking scenario, we learn that the loop consisting of the tasks Order and Confirm, the complex router Merge, and the successful condition orders=1 can deplete the condition p3 (see Figure 7.14 on page 200). The resulting state in the workflow process definition corresponds to the marking [Confirm/p1, Ship/p1], from which we can reach the state corresponding to the marking [Confirm/p1, wait/p2], after which all tasks are dead. Obviously, the workflow process definition does not enforce any synchronization between the left incoming edge and the top incoming edge of the wait. Indeed, if we would remove the wait from the workflow process definition, yielding a Staffware process definition that is shown in Figure 8.15, the resulting WF-net would be sound. Therefore, we conclude that the wait is anomalous, and the process definition needs to be updated accordingly.

### 8.2.2 Life-cycle inheritance

For checking life-cycle inheritance, we need two sound workflow process definitions. For this reason, we use the Staffware Original process as shown in Figure 7.9 on page 195 (which is mapped onto a sound workflow process definition) and the repaired Staffware Revised process as shown in Figure 8.15 (which is also mapped onto a sound workflow process definition, as we have seen in the previous subsection). Furthermore, as explained in Chapter 7, we make sure that both workflow process definition have the same number of available controls (using a text editor). After loading both nets, we open the Inherit-
ance Checker dialog, select the Original workflow process definition as base class, and the Revised workflow process definition as potential subclass, which is shown in Figure 8.16. Note that we also selected that we want to check life-cycle inheritance. After pressing the Check button, Woflan shows us that the repaired Revised workflow process definition is a subclass of the Original workflow process definition under
life-cycle inheritance, which is shown in Figure 8.17. Although this is not reflected by Figure 8.16, the backtracking algorithm needs to check branching bisimilarity once, as the only way to satisfy all constraints is to hide labels Archive and Confirm and to block label Reconnect.

8.3 Web service application

Another application that uses the libraries is a Web service application. Basically, one can send a process definition (using a known file format) to a Web server and specify whether one wants to have a “dummies report” or an “experts report”. The major difference between the dummies report and the experts report, is that the latter includes information that can only be used by knowledgeable users, like, for example, OR-AND mismatches in case of liveness problems.

Figure 8.18 shows the architecture of the Web service application. The major difference with the architecture of the Woflan 2.2 Windows application is the absence of the dynamic library wofdll.dll. Instead, the Web service application directly addresses the static library woflib.lib.

The wofcgi.exe executable is the Windows-based Woflan CGI script. Similar to the Windows-based Woflan application, this CGI script uses all static libraries as discussed in Section 8.1 to check relevant net properties. Note that this CGI script does not use the dynamic library wofdll.dll, instead it directly uses the static library wofdll.lib.
The Web service application loads the process definition (similar to the Windows application, see Figure 8.5) and generates an XML format file containing all relevant properties of the loaded process definition. This XML file contains a link to an XSL stylesheet that will transform the XML file into a report. To do so, like the Windows application, the XSL stylesheet implements the diagnosis process (see Figure 4.14 on page 97). The XML file is sent back to the client’s browser, which performs the necessary XSL transformation and shows the report.

Once again, we take the Staffware Revised process as example. Figure 8.19 shows an example Web service form for this process, after we have inserted, using copy and paste, the process definition, and after we have specified that the input is in Staffware’s XFR format and that we want to have a dummies report. When we press the Submit button, the Web service application converts, using the Staffware conversion library xfr2tpn.lib, the process definition and generates the XML file. When a browser stylesheet-aware browser is to display this XML file, it will load the corresponding XSL file, transform the XML file accordingly, and show the result, which is shown in Figure 8.20. Note that the Web service application uses the same nomenclature as the Windows application (see Table 8.1).
8.4 Ongoing work

The next release of Woflan will be Woflan 3.0. This release will extend Woflan 2.2 with the following features:

- A loader for Petri-Net Markup Language (PNML) [JKW00, BCH+03] files in the static library woflib.lib, because we want to use the well-received PNML format as Woflan’s native format instead of the proprietary TPN format.

- The backtracking algorithm as presented in Chapter 5 in the static library grcps.lib. A prototype of this algorithm has been implemented as a command-line utility for testing purposes, but this algorithm still needs to be included in Woflan.

- The invariant algorithms as presented in [CS90]. Up to Woflan 2.2, a standard equation solver has been implemented to compute invariants. We expect the (dedicated) invariant algorithms as pre-
Conclusions

8.5 Conclusions

Two applications have been build that are able to verify workflow processes defined using various workflow management systems. The first application is a Windows application, the second application is a Web service (to be precise, a CGI application). Both application use the same set of static libraries, which contain the necessary functionalities.

- A mapping library for IBM MQSeries Workflow, as has been described in Section 7.1.

FIGURE 8.20. A Web service dummies report on the Staffware Revised process definition.

presented in [CS90] to outperform the (standard) equation solver. The prototype implementing the backtracking algorithm already implements the invariants algorithms from [CS90]. Experiences with this prototype seem to suggest that our expectation holds, although we do not present any numbers here to support this.
To allow for Woflan to be used as a back-office tool, several static libraries have been combined into one dynamic library (a Windows DLL). The Windows application uses this dynamic library, but the Web service does not.

In total, five static libraries have been implemented, which should compile on any platform (they have been implemented on Windows and UNIX platforms). Two of the libraries implement mappers from Staffware, COSA Workflow, and Protos. The next two libraries implement the branching bisimilarity algorithm and the other necessary (WF-)net-related algorithms. The last library offers a simplified API to the algorithms in the former two libraries. This simplified API is also used for the dynamic library.

Furthermore, a prototype application (to be precise, a command-line utility) has been build that implements the backtracking algorithm as introduced in Chapter 5.
This chapter puts the existing releases of Woflan and the prototype implementing the backtracking algorithm, which are described in Chapter 8, to the test using three case studies. The first two case studies test to what extent existing releases of Woflan can be used to diagnose soundness, using the diagnosis process and techniques as described in Chapter 4. The first case study uses a complex Staffware process definition, the second uses a hierarchical Protos process definition. The third case study tests how the backtracking algorithm as described in Chapter 5 performs in comparison to an exhaustive search algorithm. For this, we use a Protos process definition and five successive extensions of this Protos process definition. The results of the case studies are encouraging, and in one case study, unexpected.

9.1 Staffware case

In cooperation with Staffware Benelux, we set up an experiment to test Woflan 2.1 on a real-world workflow process definition. (At the time of the case study, version 2.1 was the latest release of Woflan.) The starting point of the case study was a correct complex Staffware process definition containing 114 tasks and other Staffware building blocks (like waits, complex routers, and conditions). Figure 9.1 shows a bird’s eye view on the entire process definition, which was developed by Staffware Benelux using Staffware 2000 [Sta02].

On our request, Staffware Benelux introduced a number of typical errors in the process definition. Both the number and the type of errors were unknown to us. We diagnosed the resulting process definition
using Woflan 2.1, corrected the process definition, and discussed our diagnosis results with Staffware Benelux.

9.1.1 Diagnosing and correcting the process definition

In this section, we discuss the actual diagnosis of the Staffware model used for this case study. Table 9.1 summarizes the results. The case
Three iterations were needed for the diagnosis, taking in total about one and a half hour. Given the size of the workflow process definition and the fact that we were not familiar with the process, in our opinion, this effort is reasonable. In the first two iterations, we found and corrected six (out of seven) errors; the third iteration showed that the model resulting from the first two iterations was sound.

During the first iteration, one error was detected during the mapping (Step 1: Start of diagnosis). A small part of the process definition (see Figure 9.2) was not connected to the main part. This led to the following suggestion for a first correction:

1. Remove the unconnected part from the process definition.  

Furthermore, three structural errors were found and corrected using the mismatches reported in Step 4 (Confusions and mismatches): A complex router needed to be replaced by a wait and two links needed to be removed. The first structural error is shown in Figure 9.3: The complex router P6, which acts as an AND-split, is (partly) complemented by the router ORJOIN, acting as an XOR-join. This led to the following suggestion for a second correction:

2. Replace complex router ORJOIN by a wait.

1. Later on in the case study, Staffware confirmed that this unconnected part corresponded to one of the seven errors.
The essence of the second structural error is shown in Figure 9.4: Due to the link from complex router P16 to step 20 (Besturing & Bewaking), step 20 is improper. This led to the following suggestion for a third correction:

3. Remove the link from complex router P16 to step 20.

Likewise, the essence of the third structural error is shown in Figure 9.5: The two incoming links for complex router P27 can be tra-
versed in parallel, thus, complex router P27 is improper. This led to the following suggestion for a third correction:

4. Remove the link from the condition following step 10 (Besturing & Bewaking) to complex router P27.

After having applied these four suggestions and making the appropriate corrections, we started the second iteration. In this iteration, we did not find any more structural errors, but we did find two behavioral ones. The locking scenarios of Step 13 of the diagnosis process clearly indicated that the process contained two erroneous XOR-splits.

The first erroneous XOR-split is step 10 (Vullen NCP MP3) shown in Figure 9.3. Although this figure suggests that this step is similar to step 8 (Vullen C7 NCP MP10), the scenarios reported by Woflan indicate that it is not similar. Close inspection of the process definition, revealed that step 10 is withdrawn on expiry of its associated deadline, whereas step 8 is not. Withdrawing step 10 causes a synchronization error further on, which was signaled by locking scenarios. This led to the following suggestion for a fifth correction:

5. Set the deadline of step 10 to leave step 10 scheduled (instead of withdrawing it) when it expires.

The second erroneous XOR-split is the condition just before step 9 (Aanmaken Routepl.MP7) shown in Figure 9.6. If the condition evaluates to false, its branch terminates. In this particular case, this implies an error because further on the synchronization via the wait step following complex router P9 will fail. (This mistake might seem obvious; however, recall that the total workflow consists of over 100 building blocks, which makes it much harder to find the mistake using only visual inspection.) This led to the following suggestion for a fourth correction:

FIGURE 9.5. A link to complex router P27 needs to be removed.
6. Add a link from the bottom of the Staffware condition preceding step 9 to complex router P9.

After these suggestions were applied, the resulting Staffware process definition corresponded to a sound WF-net.

We reported the six suggestions back to Staffware, and they confirmed that these suggestions were correct, that is, the suggestions matched with six of the errors they had introduced in the process definition. However, they also mentioned that they had introduced seven errors, instead of six. The one error we failed to diagnose using Woflan was lost in the mapping. It concerns a type of error that may occur in the timeout construct of Staffware. As explained in Section 2.2, it is inherent to our approach that some errors are lost in the abstractions we apply, particularly if these errors are not or not closely related to the routing of cases. However, in this particular case, it is possible to incorporate a simple check in the mapping process to filter out this specific type of error. In fact, further experience might show that also other types of errors can be filtered out during the mapping of process definitions for use with Woflan. It is even possible that (some of) the mappings coupling Woflan with the various workflow products evolve into workflow-tool-specific extensions of Woflan for diagnosing errors that are specific for that particular workflow tool.

9.1.2 Summary

The main conclusion of this case study is that Woflan can be a useful aid for detecting and correcting errors in Staffware process definitions. In the end, we detected and corrected six out of seven errors, and the seventh error can be detected if we add a simple check. The results
Procurement case

support our belief that workflow-tool-independent verification as visualized in Figure 2.4 on page 27 is feasible.

In Section 7.2, we explained that the semantics of a Staffware procedure and its corresponding (WF-)net are not necessarily identical, and we also explained why we thought it was reasonable to diverge from the Staffware semantics. This case study showed that, although some divergence exists, we still can detect and correct any errors in the original Staffware process definition.

9.2 Procurement case

At the moment of writing this thesis, Deloitte & Touche (The Netherlands) is setting up a process repository, that is, a repository containing a substantial number of typical processes. All of these processes are modeled using Petri nets. Some of the processes are only relevant for scientists: Their main purpose is to show that Petri nets exist that have some combinations of properties (for example, a sound WF-net that is not S-coverable). Other examples are relevant for practitioners, because they model processes as they are used in the field. An example of such a process is the Procurement process we use in this case study.

The Procurement process is a hierarchical Petri net, modeled using Protos [Pal97, Wav02], containing one main process, five subprocesses, sixteen sub-subprocesses, and one sub-sub-subprocess. Figure 9.7 shows the entire hierarchy of this process as a UML class diagram. Note that in this hierarchy consists of 23 processes.

The goal of this case study is to show, using Woflan 2.2, that the entire Procurement process is sound, and, if it is not sound, to diagnose any errors and to correct them. Basically, for diagnosing a hierarchical process definition, two approaches exist:

• compositional, which starts at the top of the hierarchy and descends the hierarchy if and only if errors are detected at some level;
• incremental, which starts at the bottom of the hierarchy and ascends the hierarchy if and only if no errors are detected at some level.

We used the incremental approach, although it takes at least 23 soundness checks, whereas the compositional approach might only require one soundness check. The decision for using the incremental approach is based on our experience that sound process definitions have typically smaller state spaces than unsound process definitions. Thus, if we would use the compositional approach, then the diagnosis of the top process definition might be hindered considerably by errors in the subprocesses.
Using the incremental approach, we started with the Det. Suppliers Long List subprocess and ended with the Procurement process itself. The next subsections discuss the errors we detected in the 23 processes, of which 20 were sound. Only the process definitions Receive Materials Against P.O, Receive Goods, and Procurement (that is, the main process) contained errors.

9.2.1 Receive Materials Against P.O

Figure 9.8 shows the Protos [Pal97, Wav02] process definition of the original Receive Materials Against P.O process. In this figure, we use the symbol \( \bigtriangleup \) for routing constructs, and the symbol \( \bigtriangleup \) for tasks. Furthermore, we use *italics* to differentiate connection conditions from task, condition, and routing construct names, and we assume that connection conditions indicate XOR-splits and/or XOR-joins. Thus, for example, the task Contract applies? is an XOR-split, as its output connections have conditions.

The original Receive Materials Against P.O process is not a workflow process definition, because the routing construct Routing Join3 is useless. Close inspection of the process definition with respect to this routing construct revealed that this construct has no output conditions. To
correct this, we can either remove it, or we can add an output condition to it. Removing might result in a locking problem (which we might have to correct by adding construct Routing Join3 again), therefore, we opted for adding an output condition. Because we lacked the proper domain knowledge to decide which condition should act as output condition, we just selected one which would not cause a locking problem. This led to the following suggestion for a first correction:

1. Add a connection from routing construct Routing Join3 to place Materials rejected*.

After making this correction, we restarted the diagnosis.

The process definition corresponding to the corrected process is a workflow process definition. However, it is not coverable by threads of control: The conditions Routing_place4 and Routing_place6 are not covered. Both conditions are not covered either by any of the invariants, but they are proper. Furthermore, substates are detected, thus, the process definition cannot be sound. Therefore, there have to be non-live tasks or non-live routing constructs. Indeed, all of them turn out to be non-live, although none of them is dead. Fortunately, there is only one locking scenario:

1. Start_RMAPO
2. Perf__P_O_checks
3. Evaluate_Record_Vendor_Performance
4. Reject_Materials_bsd__on_P_O_checks_+_Materials_Rejected
5. Routing_Split2
6. Contract_Applies_+_Contract_applies
7. Routing_Join2
8. Evaluating_Contract_Perf__Mandatory_+_Evaluation_Done_

At this point, it is worthwhile to mention the fact that the Protos add-on for Woflan replaces any punctuation mark or white space by an underscore when exporting to Woflan. Thus, the task Perf. P.O checks in Protos is mapped onto a task Perf__P_O_checks in Woflan. Furthermore, Woflan 2.2 splits up any XOR-join and XOR-split to a number of AND-joins/AND-splits (transitions). To ensure that the names of these transitions are unique, relevant information concerning the input conditions (in case of an XOR-join) and/or output conditions (in case of an XOR-split) is appended to the name of the transition. As a result, the XOR-split called Reject Materials bsd. on P.O checks? in Protos is mapped to the following two transitions in Woflan 2.2:

- Reject_Materials_bsd__on_P_O_checks_+_Materials_Rejected
- Reject_Materials_bsd__on_P_O_checks_+_Materials_not_rejected.

Using Figure 9.8, it is straightforward to check that the scenario given above leads to the state where only condition Evaluation done* is true.
Procurement case

(that is, contains a token). In this state, all tasks are dead, as both Routing Join 5 and Routing Join 6 need two conditions to be true. This leads to the assumption that task Evaluating Contract Perf. Mandatory? should also make condition routing place4 true when making condition Evaluation done* true. As a result, routing construct Routing Join 6 can be executed and so forth. This leads to the following suggestion for a second correction:


After we made this correction we restarted the diagnosis process, and we found that the resulting workflow process definition was sound.

9.2.2 Receive Goods

Figure 9.9 shows the Receive Goods process definition. The hexagonal objects with symbol \(\square\) (like, for example, Receive Materials / Services) correspond to subprocesses. Note that the Receive Goods process definition contains the Receive Materials Against P.O process definition as a subprocess, which has been diagnosed and corrected in the previous section.

The corresponding net is a workflow process definition, but it is not sound due to non-live tasks or non-live routing constructs. It is also not covered by threads of control or invariants: the conditions Involved parties updated and Routing place1 are not covered by any of the sixteen threads of control or by any of the invariants. Nevertheless, again, all conditions are proper. Furthermore, again, substates do exist, all tasks and all routing constructs are non-live and non-dead, and exactly one locking scenario is reported. From this scenario we learn that if the subprocess Disposition of Inbound Material results in a new quality inspection (that is, if routing construct Rdy to perform quality inspection? is executed), condition Rdy to perform quality inspection is made true, but that condition routing place1 should also be made true (like routing construct Routing Split2 does). This leads to the following suggestion for a first correction:

1. Add a connection from routing construct Rdy to perform quality inspection? to condition routing place1.

After we made this correction, the resulting process definition was mapped onto a sound WF-net, containing 32 threads of control covering 113 conditions, and 132 tasks.
9.2.3 Procurement

Figure 9.10 shows the Procurement process definition. Note that this process definition contains the Receive goods process definition as a subprocess, which has been diagnosed and corrected in the previous section. The Procurement process definition is a workflow process definition, but it is not sound. None of the 296 conditions, including the 113 conditions from the Receive goods subprocess definition, is covered by a thread of control, and all of them are improper. Woflan reported two improper scenarios: one that ends with routing construct Supplier known? and one that ends with routing construct Supplier unknown?. From this, we concluded that the Procurement process can only avoid improper completion if routing construct Requisition not created? is executed.
In this situation, we simply played ‘the token game’, that is, we simulated the process by hand. We knew that we inevitably would run into trouble when we would execute either routing construct Supplier known? or routing construct Supplier unknown?. Therefore, we executed one of these routing constructs and observed what happened. Fortunately, because we used a bottom-up approach, we knew that all subprocesses were sound, thus, we could restrict ourselves to the main process. By playing this token game, it became evident that subprocess P_4_RG (Receive Goods) should be followed by an XOR-split modeling a choice between mtrl/serv received successfully and mtrl/serv. NOT received successfully. Instead of making both conditions true (the names of the conditions also indicate that there should be a choice), only one of them should be made true. This led to the following correction:

1. Remove the connections from subprocess P_4_RG to conditions mtrl/serv. NOT received successfully and mtrl/serv. received successfully, add a new XOR-split, add a connection from subprocess P_4_RG to the new XOR-split, and connections from the new XOR-split to conditions mtrl/serv. NOT received successfully and mtrl/serv. received successfully.

After we made this correction, the entire resulting Procurement process was mapped to a sound WF-net, containing 256 threads of control covering 297 conditions, and 367 tasks and/or routing constructs.

9.2.4 Summary

The main conclusion of this case study is that Woflan can be used to diagnose hierarchical process definitions when using a bottom-up approach, provided that every subprocess should correspond to a sound WF-net. In the entire process, we detected and corrected four errors. After making all corrections, the resulting WF-net is sound.

A top-down approach when checking soundness on a hierarchical process definitions seems to be a bad idea. If errors are detected in the main process, we still have to take the entire set of subprocesses into account when trying to diagnose the errors. For this reason, a bottom-up approach is more appropriate: When trying to diagnose errors on a higher level, at least we know that all subprocess are sound.

9.3 Transit case

This case study was handed to us by a consultant of Deloitte & Touche. In this case study, the consultant, who was knowledgeable in the area
Transit case

of soundness and life-cycle inheritance, tried to show that one can gradually add more behavior to a process definition, while preserving soundness, in such a way that every next process definition (which embeds the new behavior) is a subclass under life-cycle inheritance of the former process definition (which lacks the new behavior). First, an initial process definition was constructed, using the most typical use case for the process. Then, in five steps, an additional nineteen other use cases were added to the process definition. This resulted in a process definition that embeds the behavior of all twenty use cases. Thus, six processes were obtained: transit1 through transit6. Furthermore, a claim was made that every extension process was a subclass under life-cycle inheritance of the extended process, which made this case study a good case study for our purposes. For sake of completeness, we mention that we were given only the pictures of the process definitions, and that we first modeled these process definitions using Protos. The figures shown throughout this section are the input received from the consultant. Furthermore, this case study was performed on a Pentium 4 2.00 GHz computer with 256 Mb of RAM running Windows 2000 SP 3.

9.3.1 Performance results

Our primary goal with this case study was to see how the performance of the backtracking algorithm (BA, see Chapter 5) compares to the performance of the exhaustive search algorithm (ESA, that is, the backtracking algorithm using an empty set of constraints, see also Chapter 5). For this reason, we ran both algorithms on every process definition and its extension, using the prototype implementing the backtracking algorithm as mentioned in Chapter 8. Table 9.2 shows the results. The times mentioned in this table are the measured processing times of the algorithms to check life-cycle inheritance, that is, they exclude setup times and so on. For the remainder of this section, we identify each case by its extension process definition. Thus, the case transit4 uses the transit3 process definition as base WF-net and the transit4 process definition as potential sub WF-net.

The results for cases transit3 and transit4 were to be expected: in both cases, blocking all non-base labels is a solution. The exhaustive search algorithm always outruns the backtracking algorithm if this is the situation, as is explained in Chapter 5. The results of the other three cases seem normal and quite satisfactory. However, to our surprise, for none of these cases a life-cycle inheritance relation exists between the base WF-net and the potential sub WF-net, although this was claimed.

At this point, it is worth mentioning that, at the moment this case study was conducted, several researchers knowledgeable in the field of life-cycle inheritance had already read the case study, and none of them had...
observed that in three out of five cases the claimed life-cycle inheritance relation was absent. In our eyes, the fact that nobody observed this discrepancy, shows that one easily underestimates the subtlety of the life-cycle inheritance relation. Thus, we conclude that we need software tools to check for this relation.

9.3.2 Diagnosing and correcting the processes

Of course, knowing that the consultant’s case study would be broken if these inheritance relation were not established in all five cases, we tried to diagnose the absence of these relations, even though, at this point, Woflan and the prototype do not really support diagnosing errors related to the absence of life-cycle inheritance. Note that diagnosing and correcting these processes was not a goal of the original case study. However, for us, it is interesting to see to what extent Woflan and the prototype can be of help.

Case transit2. Figure 9.11 shows the transit1 process definition, whereas Figure 9.12 shows the transit2 process definition. Each process involves four parties who exchange messages. Basically, each party proceeds in the vertical plane, whereas each message proceeds in the horizontal plane. In general, the task S_xxx sends the message xxx, and the task R_xxx receives message xxx. Thus, the task S_dec_dat sends the message dec_dat, the task R_dec_dat receives this message.
FIGURE 9.11. The transit1 process definition.
FIGURE 9.12. The transit2 process definition.
In the first iterative step, the behavior related to erroneous message transfers between the two parties in the middle is added. Figure 9.13 shows the net fragment for error-free communication between the two middle parties, whereas Figure 9.14 shows the solution to non-error-free communication as is introduced in the transit2 case (see also Figure 9.12). According to this solution, upon reception of an erroneous arr_adv message, the recipient now may send a fun_nck message back to the sender, who has to send the arr_adv message again upon reception of this fun_nck message. This process of sending a fun_nck and resending an arr_adv keeps repeating until a correct arr_adv message is received. To prevent the sender of the arr_adv message from proceeding while the fun_nck message has not yet been received, a time_out message has been added, even though this message was not present in any of the relevant use cases. If a correct arr_adv message is received, the recipient sends a time_out message back to confirm the proper reception of the arr_adv message. Only after this confirmation can the sender of the arr_adv message proceed.

To diagnose why there exists no life-cycle inheritance relation between both processes, we examined the states that were not branching bisimilar to any other state for the situation where the labels S_time_out and R_time_out are hidden and the labels S_fun_nck and R_fun_nck are

---

**FIGURE 9.13.** Error-free communication between two parties.

**FIGURE 9.14.** First solution to non-error-free communication between two parties.
Case studies

256

blocked (which seems like a perfect candidate for a life-cycle inheritance relation, see also Figure 9.14). In the end, we discovered that the confirmation of proper reception of the arr_adv message is the cause of the absence of the life-cycle inheritance relation. In the transit1 process, the party that sends the arr_adv message can proceed to send the des_con message before the other party has received the arr_adv message. In the transit2 process, this is impossible, because the party that sends the arr_adv message has to wait until the other party has confirmed proper reception of this message.

Apparently, the solution chosen by the consultant to add the behavior related to erroneous message transfers is not compatible with the life-cycle inheritance relation. This problem is clearly due to the fact that in the use cases, and thus, in practice, the time_out message does not exist. Instead, it seems safe to assume that, in practice, the sender of the arr_adv message will wait for a certain amount of time before proceeding. If the sender does not receive a fun_nck message during that period of time, he will proceed. Fortunately, there is a way to capture this behavior, which is shown in Figure 9.15: The sender will wait for a fun_nck message as long as condition p holds. After this period of time, that is, when condition p does not hold anymore, the party that receives the arr_adv message cannot send a fun_nck anymore. Note that this solution makes the time_out message obsolete, which is a point in favor, because this message was not a part of the use cases this extension was based on. This led to our first suggestion for correcting the absence of a life-cycle inheritance relation in the transit2 case:

1. Relabel the tasks related to the time_out message to τ (thus, hide these tasks), remove the place in-between S_time_out and R_time_out tasks, remove the connection from place p to task R_fun_nck, and add a connection from place p to task S_fun_nck (as is shown in Figure 9.15).
Transit case

**Case transit5.** Figure 9.16 shows the transit4 process definition, whereas Figure 9.17 shows the transit5 process definition. In the transit5 process definition, a dummy task has been added (labeled GO), and tasks have been added for receiving the message arr_not, for sending the message time_out, and for sending and receiving the messages lar_req and lar_rsp.

We cannot hide or block the new tasks that receive the message arr_not or send the message time_out, because tasks with the same label are present in Figure 9.16. As a result, in the transit5 process, we can send a time_out message immediately after having send a mrn_all message. However, in the transit4 process, we cannot do this. Thus, no matter how we hide and/or block the new labels (lar_req and lar_rsp), both processes will behave differently. As a result, the transit5 process definition is not a subclass under life-cycle inheritance of the transit4 process definition.

Recall that in the previous subsection we already pointed out that the time_out message is obsolete for the extension from process transit1 to process transit2. If we would remove the tasks sending and receiving these messages in process transit2 (and, hence, from the processes transit3, transit4, transit5, and transit6), the label S_time_out would not exist in the transit4 process, and, hence, we could hide or block it. Thus, applying the suggestion for correcting the previous error might also correct this one. This led to the following suggestion to correct the absence of a life-cycle inheritance relation in the transit5 case:

1. Apply the suggestion for correcting the absence of a life-cycle inheritance relation in the transit2 case.

**Case transit6.** Figure 9.18 shows the transit6 process definition. The transit6 process definition extends the transit5 process definition (see Figure 9.17) with a lot of new tasks and routing constructs. However, we focus on the two new routing constructs in the dashed area in Figure 9.18.

Both routing constructs will be hidden, because we only want to observe tasks. Furthermore, because the transit6 process definition is sound, both routing constructs can be executed. However, executing any of the routing constructs results in two tokens: one for an R_can_dec task and one for an S_can_not task, which are both new. Because input places of these tasks are markable (see also Section 5.3), these tasks may not be blocked. But, if these tasks may not be blocked, then the new tasks S_can_trans and R_can_not may not be blocked too, and so on. In the end, none of the new tasks S_can_dec, S_can_trans, R_can_trans, S_can_not, R_can_not, S_can_ack, and R_can_ack may be blocked, and, hence, must be hidden. But hiding
FIGURE 9.16. The transit4 process definition.
FIGURE 9.17. The transit5 process definition.
FIGURE 9.18. The transit6 process definition.
these new tasks introduces new alternative paths, which introduce new behavior. For example, in the transit5 process definition a task $R_{\text{delay}}$ must eventually be followed by a task $S_{\text{sta_req}}$, but, after hiding all the new tasks mentioned above, this needs not be the case for the transit6 process definition.

This leads to the conviction that, when modeling the transit6 process definition using Protos, we misinterpreted the text $S_{\text{can_dec}}$ for a condition’s name, where it was meant as a label for both tasks, and thus to the following suggestion for correcting the absence of a life-cycle inheritance relation in the transit6 case:

1. Label the tasks we believed to be unlabeled (and, thus, routing constructs) with the label $S_{\text{can_dec}}$.

After we applied the above mentioned suggestions and corrected the process definitions accordingly, the life-cycle inheritance relation was present in all five cases.

### 9.3.3 Performance results on the corrected processes

On the corrected process definitions, we again ran a comparison between the performance of the backtracking algorithm and the performance of the exhaustive search algorithm. Table 9.3 shows the results. The results for the transit2, transit3, transit4, and transit6 case can be

<table>
<thead>
<tr>
<th>Base WF-net</th>
<th>transit1</th>
<th>transit2</th>
<th>transit3</th>
<th>transit4</th>
<th>transit5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Places</td>
<td>39</td>
<td>47</td>
<td>51</td>
<td>65</td>
<td>72</td>
</tr>
<tr>
<td>Transitions</td>
<td>24</td>
<td>35</td>
<td>39</td>
<td>51</td>
<td>58</td>
</tr>
<tr>
<td>Reachable states</td>
<td>77</td>
<td>156</td>
<td>171</td>
<td>214</td>
<td>242</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Potential sub WF-net</th>
<th>transit2</th>
<th>transit3</th>
<th>transit4</th>
<th>transit5</th>
<th>transit6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Places</td>
<td>47</td>
<td>51</td>
<td>65</td>
<td>72</td>
<td>96</td>
</tr>
<tr>
<td>Transitions</td>
<td>35</td>
<td>39</td>
<td>51</td>
<td>58</td>
<td>82</td>
</tr>
<tr>
<td>Reachable states</td>
<td>156</td>
<td>171</td>
<td>214</td>
<td>242</td>
<td>338</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Life-cycle inheritance?</th>
<th>Yes</th>
<th>Yes</th>
<th>Yes</th>
<th>Yes</th>
<th>Yes</th>
</tr>
</thead>
</table>

**TABLE 9.3. Performance results of the corrected transit case.**

<table>
<thead>
<tr>
<th>Time: BA (in seconds) w/ 99% conf. interval</th>
<th>transit1</th>
<th>transit2</th>
<th>transit3</th>
<th>transit4</th>
<th>transit5</th>
<th>transit6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Places</td>
<td>$1.34\times10^{-2}$</td>
<td>$3.68\times10^{-2}$</td>
<td>$4.78\times10^{-2}$</td>
<td>$9.79\times10^{-2}$</td>
<td>$1.25\times10^{-1}$</td>
<td>$1.19\times10^{-1}$</td>
</tr>
<tr>
<td>Transitions</td>
<td>$\pm4.45\times10^{-4}$</td>
<td>$\pm4.13\times10^{-3}$</td>
<td>$\pm7.16\times10^{-4}$</td>
<td>$\pm7.17\times10^{-4}$</td>
<td>$\pm5.01\times10^{-4}$</td>
<td>$\pm4.06\times10^{-4}$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Time: ESA (in seconds) w/ 99% conf. interval</th>
<th>transit1</th>
<th>transit2</th>
<th>transit3</th>
<th>transit4</th>
<th>transit5</th>
<th>transit6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Places</td>
<td>$1.25\times10^{-2}$</td>
<td>$3.44\times10^{-2}$</td>
<td>$4.66\times10^{-2}$</td>
<td>$1.92\times10^{-1}$</td>
<td>$1.19\times10^{-1}$</td>
<td>$1.19\times10^{-1}$</td>
</tr>
<tr>
<td>Transitions</td>
<td>$\pm4.50\times10^{-4}$</td>
<td>$\pm1.78\times10^{-4}$</td>
<td>$\pm3.41\times10^{-4}$</td>
<td>$\pm1.64\times10^{-3}$</td>
<td>$\pm6.54\times10^{-4}$</td>
<td>$\pm4.06\times10^{-4}$</td>
</tr>
</tbody>
</table>

261
explained by the fact that blocking all non-base labels is a solution. As mentioned before, the exhaustive search algorithm always outperforms the backtracking algorithm if this is the case, because it does not have to compute the constraints.

In the transit5 case, the backtracking algorithm clearly outperforms the exhaustive search algorithm, although the latter algorithm only needs to check branching bisimilarity twice: Hiding non-base label GO and blocking the other non-base labels is a solution, and, by chance, label GO is at the bottom level in our search tree. Thus, after the branching bisimilarity check with all non-base labels blocked fails, the next check is with label GO hidden and all others blocked. After we repositioned label GO at the middle of the tree, the exhaustive search algorithm took $4.85 \times 10^{-1} \pm 1.05 \times 10^{-3}$ seconds, after we repositioned it at the top, it took $3.17 \times 10^6 \pm 1.17 \times 10^{-2}$ seconds. An extreme example illustrating that the position of labels in the search tree can have a dramatic effect on the processing time, is the case when using process transit1 as base WF-net and process transit6 as potential sub WF-net. For this case, the backtracking algorithm took $2.73 \times 10^{-2} \pm 2.10 \times 10^{-4}$ (label GO at the bottom), $2.71 \times 10^{-2} \pm 2.01 \times 10^{-4}$ (at the middle), and $2.71 \times 10^{-2} \pm 3.25 \times 10^{-4}$ (at the top) seconds, but the exhaustive search algorithm took $3.19 \times 10^{-2} \pm 1.78 \times 10^{-4}$ (at the bottom), $1.45 \times 10^{2} \pm 2.78 \times 10^{-1}$ (at the middle), or approximately $2.38 \times 10^{6}$ (at the top) seconds (that is, almost four weeks). We did not measure the latter number, because it simply takes too much time; instead, we extrapolated the previous result in the following way:

- The search tree contains 28 levels, where level 1 is the top level, level 15 is the middle level, and level 28 is the bottom level;

- Thus, finding the solution when label GO is positioned at the middle takes $8193 \left(2^{28-15} + 1 \right)$ branching bisimilarity checks, which took approximately 145 seconds;

- Because finding a solution when label GO is positioned at the top takes $134,217,729 \left(2^{28-1} + 1 \right)$ branching bisimilarity checks, this will take approximately $2.38 \times 10^{6} \left(\frac{\left(2^{27} + 1 \right)}{\left(2^{13} + 1 \right)} \right) \times 145$ seconds.

- Note that we assume that the branching bisimilarity check is the dominant factor with regard to the processing time and that the branching bisimilarity checks, when label GO is positioned in the middle, are representative for all these checks.

Thus, were the processing time of the exhaustive search algorithm reaches an excessive level when label GO is positioned near the top, the positioning of this label seems to have (almost) no effect on the processing time of the backtracking algorithm. As a result, the backtracking algorithm reduces the excessive processing times of almost four weeks to a fraction of a second.
9.3.4 Summary

The primary conclusion of this case study is that the backtracking algorithm can prevent excessive processing times of the exhaustive search algorithm by orders of magnitudes (for example, a fraction of a second instead of almost four weeks). In contrast with this, the exhaustive search algorithm might outrun the backtracking algorithms too, but usually this is only the case when blocking all non-base labels yields a life-cycle inheritance relation, and the difference is clearly within limits and acceptable. Furthermore, our experience with the backtracking algorithm suggests that it needs only one branching bisimilarity check if a life-cycle inheritance relation is present.

We can also try to control the exhaustive search algorithm by using intermediate steps, as this case study shows. As mentioned, when trying to check for life-cycle inheritance between processes transit1 and transit6 in case label GO is at the top level of the search tree, the exhaustive search algorithm takes about four weeks to compute. However, using 4 intermediate steps, the combination of five exhaustive search algorithms takes approximately $3.29 (0.015 + 0.0344 + 0.0466 + 3.17 + 0.119)$ seconds. Thus, divide and conquer might also be a good technique to lessen performance problems with the exhaustive search algorithm. However, this may not always be possible.

Another conclusion is that, although we did it for different reasons (see Chapter 5), it seemed to be a wise decision to try to block a non-base label before we try to hide it: It seems that, in general, the majority of the non-base labels needs to be blocked. (Note that the case study performed in Chapter 5 also supports this conclusion).

A third conclusion is that we really need a software tool to check whether a life-cycle inheritance relation exists between two process definitions. In this case study, a number of knowledgeable people did not detect that in certain cases the claimed life-cycle inheritance relation was absent. Only after we checked this with Woflan, this became apparent.

At the moment, Woflan does not provide any diagnostic information related to life-cycle inheritance, except perhaps for a branching bisimulation relation in case of a seemingly perfect hiding and blocking scheme. Using such a scheme, we were able to correct one absent life-cycle inheritance relation, but it took considerable effort. The other two cases in which the life-cycle relation was absent were diagnosed by simply accepting the fact that this relation was absent and to look for possible causes. However, it would be nice if Woflan could give some guidance when trying to diagnose the absence of a life-cycle inheritance relation. A possible idea would be to have the designer of the pro-
cess definitions specify a hiding and blocking scheme, and to report back (in some way) the boundary between branching bisimilar states and non-branching bisimilar states. By definition, the successful terminal states are branching bisimilar, and, obviously, the initial states are not. Somewhere between the initial states and the successful terminal states a boundary exists that separates branching bisimilar states from non-branching bisimilar states. Note that, by definition, arcs can only cross this boundary from non-branching bisimilar states to branching bisimilar states. Apparently, for some reason, on this boundary the branching bisimulation gets lost, which makes it very interesting from a diagnosis point of view.

9.4 Conclusions

In general, we can state that the Woflan tool can be used in an effective and efficient way to diagnose process definitions. Using the approach shown in Figure 2.4 on page 27 we can, in an iterative manner, diagnose and correct a process definition. The techniques Woflan offers to diagnose unsoundness seem to be sufficient. However, Woflan still lacks proper techniques when trying to diagnose the absence of a life-cycle inheritance relation.

Finally, during the case studies, we have noticed that checking for soundness or for a life-cycle inheritance relation typically takes considerable more time when the property is absent. As a result, when it takes Woflan considerable time to compute one of these properties, it seems safe to assume that the property is absent.
To conclude this thesis, this chapter lists its main contributions, the limitations of the results we obtained, and possible future work.

10.1 Contributions

As has been indicated in Chapter 1, this thesis aims to build a bridge between the current workflow management systems and state-of-the-art verification techniques. As pillars of the proposed bridge, this thesis introduces a diagnosis process for soundness, behavioral error sequences to diagnose soundness-related errors, a backtracking algorithm that eases excessive processing times in the context of life-cycle inheritance, a number of mappings from existing process definition formats to WF-nets, and a number of case studies.

10.1.1 Diagnosis process and behavioral error sequences

Chapter 4 has presented a diagnosis process that exploits the fact that a WF-net can only be sound if its short-circuited net is live and bounded (see Theorem 3.2 on page 66). Therefore, instead of checking the three soundness requirements, the diagnosis process checks the well-known liveness and boundedness properties for the short-circuited net. Errors violating either this liveness or boundedness are reported back to the designer.

As a last resort in the diagnosis process, Chapter 4 has also introduced behavioral error sequences. Basically, a behavioral error sequence pinpoints some decision in the workflow process definition that ultimately...
will result in erroneous behavior. Thus, a behavioral error sequence pinpoints the apparent origin of an error instead of the error itself. Two classes of behavioral error sequences are introduced that help the designer to diagnose and correct errors related to liveness and boundedness: non-live sequences and unbounded sequences. Given a bounded system that contains no dead transitions, a non-live sequence pinpoints a decision in the workflow process definition which prevents the system from reaching the successful terminal state henceforth, while the successful terminal state was reachable before the decision was taken. Thus, this decision violates the liveness property and, hence, the soundness property. In a similar way, an unbounded sequence pinpoints a decision in the workflow process definition which ultimately leads to unbounded behavior. Thus, this decision violates the boundedness property and, hence, the soundness property.

Using a case study, we have shown that the diagnosis process and the behavioral sequences are useful when diagnosing and correcting a faulty process definition.

### 10.1.2 Backtracking algorithm

Chapter 5 has introduced a backtracking algorithm for computing lifecycle inheritance that eases the problem of excessive processing times to a large extent.

In one of the case studies in Chapter 9, the backtracking algorithm needed only a fraction of a second, whereas we estimated that the exhaustive search algorithm would need almost four weeks. Nevertheless, the backtracking algorithm might also perform slightly worse than the exhaustive search algorithm due to the additional overhead of computing the necessary structural properties. However, our case studies suggest that this decrease in performance is always within acceptable limits (seconds instead of weeks), which is to be expected as structural properties are in general less complex to compute than behavioral properties like branching bisimilarity checks. Our case studies also show that the exhaustive search algorithm typically outperforms the backtracking algorithm when there exists a protocol inheritance relation between both process definitions: If a protocol inheritance relation exists, then both algorithms need only a single branching bisimilarity check, but the backtracking algorithm suffers from the additional overhead.

In the context of inter-organizational workflows, a workflow typically consists of a workflow process definition for every party involved. In such a context, the backtracking algorithm can be used to decide whether a party’s public workflow matches its private workflow, that is, whether its private workflow is an abstracted view on its public.
Contributions

If this holds for all the parties involved, then we can check whether the combination of private workflows results in a sound workflow process. If so, then the combination of public workflows also results in a sound workflow process and some minimal requirements for the inter-organizational workflow are thus met. Furthermore, the backtracking algorithm can be used to decide whether or not cases can be successfully migrated from one workflow process definition to another workflow process definition. If the second workflow process definition is a subclass under lifecycle inheritance of the first, then all cases can be migrated safely, because every state in the first workflow corresponds to some state in the second.

10.1.3 Mappings

The diagnosis process as introduced in Chapter 4 is based on workflow process definitions that are mapped onto general WF-nets. For this reason, Chapters 6 and 7 have presented mappings from a number of existing process definition formats (for IBM MQSeries Workflow, Staffware, COSA Workflow, Protos, and XRL) to WF-nets. However, most likely, workflow process definitions of a certain workflow management system are typically mapped onto specific WF-nets that satisfy certain requirements by definition. For example, IBM MQSeries process definitions are mapped onto WF-nets that satisfy both the option-to-complete requirement and the proper-completion requirement. As a result, the diagnosis process can be simplified (‘tuned’) towards such a workflow management system. Chapter 7 presents results on how the diagnosis process can be simplified for the mappings introduced (see, for example, Table 7.1 on page 216.)

10.1.4 Woflan

Finally, we have presented the tool Woflan, together with its architecture, its appearances, and the design decisions involved. The tool Woflan consists of a core library containing algorithms to compute both the generic Petri-net-related and the specific WF-net-related properties, a number of libraries to map workflow process definitions to WF-nets, and a user interface that implements the diagnosis process. At the moment, Woflan supports the mapping of Staffware, COSA, Protos, XRL and Meteor workflow process definitions, and it supports both a graphical (Windows) user interface and a web-based (CGI: Common Gateway Interface) interface.

10.1.5 Case studies

We can use Woflan in an effective way to verify, diagnose, and correct real-life workflow process definitions, as a number of case studies has
shown. The first case study has shown that we were able to correct six out of seven errors in a complex Staffware process definition, the second has shown that we were able to correct a hierarchical process definition containing more than 20 subprocesses, and the third has shown that our backtracking algorithm for deciding life-cycle inheritance can reduce excessive processing time from weeks to seconds, while keeping possible increases in processing times within seconds. Thus, Woflan fulfills the promise we have made in Chapter 1 to a large extent. However, the case studies and our experiences with Woflan indicate that some work still remains to be done.

10.2 Limitations and future work

Although the presented bridge between current workflow management systems and state-of-the-art verification techniques is usable, it has its limitations and shortcomings. As a result, the proposed bridge also has its weaknesses, which are related to the exclusive focus on the control-flow perspective, the use of WF-nets, the exclusive focus on soundness as correctness criterion, the inability to provide diagnostic information related to life-cycle-inheritance related errors, and the resulting tool Woflan itself.

10.2.1 Control-flow perspective

Chapter 2 has argued that we can restrict ourselves to the control-flow perspective of workflow process definitions. Most importantly, we have argued that we can abstract from the data perspective. However, this abstraction may result in additional behavior if different decisions are dependent because they share some data elements. Chapter 2 has argued that we should not rely on data elements for two reasons:

1. Data is volatile, and might be changed in-between dependent decisions.
2. Sharing data elements results in implicit relations. Preferably, these relations should be made explicit in the workflow process definition.

However, as the remaining chapters have shown, existing workflow management systems often have to rely on data elements for dependent choices, because their control-flow language lacks the expressive power to model this in another way. In a workflow process definition with data-dependent choices, the correct unwinding of a case might depend on the actual values of the corresponding data elements. Although we feel that such a workflow process definition should be diagnosed and corrected (whatever happens to the data, any case should unwind correctly), it would be interesting to explore to what
extent we can include the data perspective in our approach. For example, we might be able to use a weaker notion of soundness that takes data elements into account.

10.2.2 WF-nets

Based on our decision that we could restrict ourselves to the control-flow perspective, we have decided to use a subclass of Petri nets called WF-nets [Aal98a] to capture the essential behavior of workflow process definitions. As a result of this decision, we immediately could benefit from over 40 years of Petri-net-based research.

Although we can successfully map process definitions from three different WFMSs (IBM MQSeries Workflow, Staffware, and COSA Workflow) and a BPR tool (Protos) onto WF-nets, some of the existing workflow management systems necessarily rely on data elements to prevent undesired behavior, as the previous section has indicated. We have shown that IBM MQSeries Workflow and Staffware are examples of such workflow management systems. Therefore, we would like to include the data perspective into their mappings. As WF-nets do not allow for data elements, this would require an extensive analysis of the use of data elements, and map this use of data elements onto transitions, places, and arcs. Figure 10.1 shows, for example, how we could incorporate the data element ‘order’ into the mapped Revised Staffware process definition (see Figure 7.15 on page 201): We add two places that symbolize the different values for the data element, and connect these places to appropriate transitions using appropriate arcs. However, how this WF-net can be generated in an automatic way, is not clear yet.

A possible alternative for WF-nets might be to use colored Petri nets, which explicitly allow for data elements. For example, we could use CPN Tools [RWL+03], which can generate and query state spaces of colored Petri nets. Standard queries on the state space include boundedness and liveness related queries, but one can also build own queries.

Another possible alternative might be a process algebra, like, for example, μCRL [BFG+01, GR01]. As μCRL is designed to support data, it might be more straightforward to map a workflow process definition onto a μCRL process. Using a model checker, we could then check whether the three soundness requirements hold for this process. Of course, this would require the soundness requirements to be mapped onto μCRL. Furthermore, to be able to use the backtracking algorithm as has been presented by Chapter 5, we would have to be able to find constraints of equivalent value in μCRL processes.
Apart from the data-perspective issue, the implemented Staffware mapping to WF-nets also poses two additional problems: completion detection and an anomalous join semantics for some objects. We could map a Staffware process definition onto a WF-net that matches the behavior of that Staffware process definition more accurately. For example, we could map a step object onto a net fragment as shown in Figure 10.2. In this figure, the net fragment corresponding to a step object can have any of the four states of a step object (see Section 7.2 on page 193). Transition \( t_1 \), \( t_2 \), and \( t_3 \) symbolize scheduling the step object, transition \( t_4 \) symbolizes releasing it (hence the observable label), whereas transition \( t_5 \) symbolizes withdrawing it. Initially, place \( p_1 \) is marked, and an instance can only complete if no step is in the state outstanding, that is, if no place \( p_2 \) is marked. Using this net fragment, it seems straightforward to map the other objects onto net fragments, for example, a net fragment corresponding to wait object needs...
the places $p_3$ corresponding to all its stop objects to be marked. However, as indicated in Section 7.2, the question is whether we want to use such a mapping, because a resulting WF-net might be sound even if it behaves in an undesired way.

10.2.3 Soundness

Recall that Chapter 1 has introduced soundness as a combination of the following three requirements:

- Option to complete, that is, a case should always be able to complete.
- Proper completion, that is, completion of a case is always proper.
- No dead tasks, that is, every task can be executed for some case.

Likewise, recall that Chapter 3 has formalized the concept of soundness on WF-nets, as follows:

- From every reachable marking the sink place can be marked.
- The only marking that marks the sink place is the successful terminal state.
- No transition is dead.

Given both third requirements, this formalization assumes that every transition in the WF-net corresponds to a task. However, as Chapter 6 and Chapter 7 have shown, this is not always the case. In these chapters, many of the transitions (every hidden transition, that is, every transition labeled $\tau$) are inserted for routing purposes. As a result, the assumption mentioned above does not hold for many workflow mappings: In practice, we only need to verify that no observable transition is dead, instead of every transition. However, if we change the formal third requirement accordingly, then we lose the relation between soundness and liveness and boundedness (see Theorem 3.2 on
Concluding Remarks

page 66), and we have to change the entire outcome of this theorem.

For example, the diagnosis process as introduced in Chapter 4 needs to be changed, and a lot of results in Chapter 4 need to be reevaluated. Another approach might be to allow our diagnosis process to remove (or block) any dead hidden transition after having issued a warning for this transition. Because the transitions are hidden, the workflow process designer is not interested in observing them. Most likely, s/he is also not interested in not-observing them.

During several case studies, we detected a tendency in ourselves to reorder the diagnosis process. Often, we first wanted to try to use the error sequences. Only if these error sequences took a long time to compute, we wanted to revert to other diagnostic information. Apparently, the fact that error sequences provide accessible diagnostic information, might be a reason to start the diagnosis process with providing these sequences, even though the complexity to compute them is bad. We expect that especially workflow process designers that don’t have any background on formal verification will have to use the error sequences to diagnose an unsound process definition.

During the last years, several variants on soundness have been coined in workflow literature. For example, Dehnert has introduced the concept of relaxed soundness [Deh03], and Van Hee et al. [HSV03] have introduced a generalized concept of soundness. Dehnert’s relaxed soundness requires that every transition can be executed for some case in such a way that proper completion can be achieved. It is straightforward to check that soundness implies relaxed soundness, but not vice versa. As a result, relaxed soundness is a weaker requirement than soundness. The reason for Dehnert to introduce the concept of relaxed soundness, is that the soundness requirements are too strict in the initial phase of defining a workflow process definition. Dehnert claims that in this initial phase, the designer is focused on modeling desired behavior (relaxed soundness), and that undesired behavior is ruled out (soundness) at a later stage. Van Hee et al. have introduced a generalized notion of soundness. Basically, they have introduced the concept of $k$-soundness, which requires that a set of $k$ cases has always the option to complete properly (where the successful terminal state is $k$ tokens in the sink place). Except for the no-dead-transitions requirement, our notion of soundness corresponds to their 1-soundness. Furthermore, their notion of soundness corresponds to $k$-soundness, for any $k > 0$. It is straightforward to show that we can map the up-to-$k$-soundness concept, for any $k > 0$, to our soundness concept: Figure 10.3 shows how this can be done. However, as far as we know, we cannot map the generalized soundness concept ($k$-soundness, for any $k > 0$) onto our soundness concept.
Chapter 9 has indicated that we do not present any diagnostic information when the check for an inheritance relation fails. It would be extremely helpful if Woflan could give the designer some feedback about why this check fails. An idea for possible diagnostic information has been given by Chapter 9. This idea exploits the fact that the successful terminal states of two sound WF-nets are always branching bisimilar, and that successors of branching bisimilar states are branching bisimilar too. As a result, if two sound WF-nets are not related through some inheritance relation, that is, if their initial states are not branching bisimilar, then we can partition the combined state space of both nets into four subsets (I, II, III, and IV) such that

- subsets I and III partition the reachability graph of the first net,
- subsets II and IV partition the reachability graph of the second net,
- subset I contains the initial state of the first net,
- subset II contains the initial state of the second net,
- subset III contains the successful terminal state of the first net,
- subset IV contains the successful terminal state of the second net,
- states in subset I cannot be reached from states in the subset III,
- states in subset II cannot be reached from states in the subset IV,
- none of the states in subset I is branching bisimilar to any of the states in the other subsets,
- none of the states in subset II is branching bisimilar to any of the states in the other subsets,
- all states in subset III are branching bisimilar to some of the states in subset IV, and
- all states in subset IV are branching bisimilar to some of the states in subset III.

**FIGURE 10.3. Checking up-to-$k$-soundness using Woflan.**
A useful metaphor in this context might be a zipper, as is shown by Figure 10.4. The reachability graph of the first WF-net corresponds to the left-hand side of the zipper, the reachability graph of the second WF-net to the right-hand side, the states corresponds to teeth, subsets I and II correspond to the separated teeth, and subsets III and IV corresponds to the fastened teeth. Our goal is to fasten the two uppermost teeth, that is, to achieve a branching bisimilarity between the initial states of both WF-nets. However, the zipper’s tag got stuck somewhere in the middle. Usually, when a zipper gets stuck, one inspects the tag to
see why. Therefore, it might make sense to inspect the boundary between subsets I and II on the one side and subsets III and IV on the other side, because this boundary corresponds to the actual position of the tag.

Another approach for diagnosing the absence of a life-cycle inheritance relation could be to use the structural information the backtracking algorithm is based on: constraints and transition invariants. By presenting the derived constraints and invariants to the designer, s/he might be able to tell that certain constraints are not as s/he expected. In case of unexpected constraints that are based on transition invariants, it might be helpful to present these transition invariants as well, as some transition invariants might be unexpected as well (and it is impossible to derive the transition invariants from the constraints only). Using this structural information, the designer might be able to detect a flaw, or why there exists no life-cycle inheritance relation between the two process definitions. For example, if some transition invariant in one process definition cannot be matched by any of the transition invariants in the other process definition, then this clearly gives an indication why the life-cycle inheritance relation is absent.

10.2.5 Woflan

Chapter 9 has mentioned briefly that there is not yet a released version of Woflan available that contains the backtracking algorithm described in Chapter 5. The main reason for this is that we are working on a complete overhaul of the existing code. We motivate the need for such an overhaul using the computation of invariants. For the backtracking algorithm as has been presented by Chapter 5, we need transition invariants to be computed as fast as possible. However, versions 1.0 to 2.2 of Woflan all use a generic equation solver for generating the relevant invariants, which is considerably slower than dedicated algorithms like the one described in [CS90]. Therefore, we decided to use the algorithm from [CS90] in the next version of Woflan, that is, Woflan 3.0.

Furthermore, we need to evaluate the use of the minimal coverability graph. Although minimal coverability graphs are typically smaller than ordinary coverability graphs, this does not necessarily mean that minimal coverability graphs can be constructed faster. During construction of a minimal coverability graph, we need to search the entire graph constructed-so-far for comparable markings, that is, for markings that are either submarkings or supermarkings. At the moment, we are unaware of any feasible search method other than an exhaustive search on the entire state space as constructed so far. In contrast, when constructing an ordinary coverability graph, we only need to check:
Concluding Remarks

- if an identical marking exists in the graph constructed-so-far, and
- if a submarking exists on some distinct path from the initial state to the current state.

For the first check, we can use a search tree based on a total order of the markings, while for the second check we only need to check the markings on this distinct path. (Note that such a total order is not possible when searching for submarkings, as two markings need not be comparable.) As a result, the ordinary-coverability-graph construction algorithm will outperform the minimal-coverability-graph construction algorithm unless during the latter algorithm a substantial part of the graph can be pruned, that is, if a lot of submarkings are detected. Thus, in case of a complex and sound workflow process definition, in which case no submarkings exist, the former algorithm definitely outperforms the latter.

At the moment, Woflan can only be used as an independent tool. As a result, a designer needs to export and/or import a workflow process definition into Woflan, and needs to interpret the diagnostic results in terms of WF-nets. From a designer’s point of view, it would be more convenient if Woflan would be integrated in the native workflow management system s/he is using: The export and/or import would be automatic, and s/he could interpret the diagnostic results in the workflow management system itself.

Using Woflan, it is not possible to check more sophisticated, and more domain-specific, properties, like, for example, the following properties for the requisition process as Chapter 1 has described:

- Every requisition request is either approved or rejected.
- For every approved requisition request the requestor receives at least one notification of forward.
- If an approver needs to be reminded, then the request will be rejected.

To answer this kind of questions, it might be worthwhile to combine our verification approach with a model checking approach. One might even consider including the data perspective when trying to answer this kind of questions, that is, one might consider using, for example, the process algebra µCRL [GR01], as µCRL is specifically designed to handle data. For instance, one could think of the following procedure:

1. Map a workflow process definitions to a UML Activity Diagram (or to a Petri net augmented with Event-Condition-Action-rules (ECA-rules)) [Esh02],
2. verify its soundness using our approach, and if it is sound (and thus corresponds to a finite transition system)
3. perform model checking using µCRL [GR01].
This appendix explains all possible invocations of the function \textit{WoflanInfo} as it is exported by the library wofdll.lib.

\begin{verbatim}
const char *WoflanInfo(
        unsigned theNet,
        unsigned theInfo,
        unsigned theIndex = 0,
        unsigned theSubIndex = 0
    );
\end{verbatim}

The function \textit{WoflanInfo} expects up to four arguments, of which the first two are mandatory. The first argument should be a handle to some loaded net, while the second argument should be some number indicating the kind of information is requested about that net. For this second argument, a number of constants has been defined, like for example, \texttt{INFO\_NAME}, \texttt{INFO\_NOF\_P}, and \texttt{INFO\_NOF\_T}. If, for example, the name of some place is requested, then the third argument should be the index to that place. Likewise, if the name of some place in some place invariant is requested, then the third argument should be the index to that place invariant and the fourth argument should be the index in that place invariant to that place.

For the second argument, the following constants have been defined. Assume that the variable \texttt{h} is a handle to a loaded net \( N = (P, T, F, I) \) and that the indices \( i \) and \( j \) are valid. If either one of the indices is invalid, any call returns the NULL-pointer. Furthermore, note that, internally, Woflan often uses arrays to represent sets. Finally, although the function is required to return a string, in many cases we assume that...
it returns a number. For sake of completeness we mention that in these cases the function returns a string containing that number.

\section{Net}

- The call \texttt{WoflanInfo}(\texttt{h}, \texttt{INFO\_NAME}) returns the name of net \( N \).
- The call \texttt{WoflanInfo}(\texttt{h}, \texttt{INFO\_NOF\_P}) returns the number of places in net \( N \).
- The call \texttt{WoflanInfo}(\texttt{h}, \texttt{INFO\_NOF\_T}) returns the number of transitions in net \( N \).
- The call \texttt{WoflanInfo}(\texttt{h}, \texttt{INFO\_NOF\_C}) returns the number of arcs in the net \( N \).
- The call \texttt{WoflanInfo}(\texttt{h}, \texttt{INFO\_P\_NAME}, \( i \)) returns the name of the \( i \)-th place.
- The call \texttt{WoflanInfo}(\texttt{h}, \texttt{INFO\_T\_NAME}, \( i \)) returns the name of the \( i \)-th transition.
- The call \texttt{WoflanInfo}(\texttt{h}, \texttt{INFO\_NOF\_PRE\_P}, \( i \)) returns the size of the preset of the \( i \)-th place.
- The call \texttt{WoflanInfo}(\texttt{h}, \texttt{INFO\_NOF\_POST\_P}, \( i \), \( j \)) returns the name of the \( j \)-th transition in the preset of the \( i \)-th place.
- The call \texttt{WoflanInfo}(\texttt{h}, \texttt{INFO\_NOF\_POST\_P}, \( i \), \( j \)) returns the name of the \( j \)-th transition in the postset of the \( i \)-th place.

\section{WF-net}

- The call \texttt{WoflanInfo}(\texttt{h}, \texttt{INFO\_NOF\_SRC\_P}) returns the number of source places.
- The call \texttt{WoflanInfo}(\texttt{h}, \texttt{INFO\_NOF\_SNK\_P}) returns the number of sink places.
- The call \texttt{WoflanInfo}(\texttt{h}, \texttt{INFO\_NOF\_SRC\_T}) returns the number of source transitions.
- The call \texttt{WoflanInfo}(\texttt{h}, \texttt{INFO\_NOF\_SNK\_T}) returns the number of sink transitions.
- The call \texttt{WoflanInfo}(\texttt{h}, \texttt{INFO\_NOF\_UNC\_N}) returns the number of nodes that are not connected to any source place or to any sink place.
The call WoflanInfo\((h, \text{INFO\_NOF\_SNC\_N})\) returns the number of nodes that are not strongly connected to any source place or to any sink place.

- The call WoflanInfo\((h, \text{INFO\_SRC\_P\_NAME}, i)\) returns the name of the \(i\)-th source place.
- The call WoflanInfo\((h, \text{INFO\_SNK\_P\_NAME}, i)\) returns the name of the \(i\)-th sink place.
- The call WoflanInfo\((h, \text{INFO\_SRC\_T\_NAME}, i)\) returns the name of the \(i\)-th source transition.
- The call WoflanInfo\((h, \text{INFO\_SNK\_T\_NAME}, i)\) returns the name of the \(i\)-th sink transition.
- The call WoflanInfo\((h, \text{INFO\_UNC\_N\_NAME}, i)\) returns the name of the \(i\)-th node that is not connected to a source place or a sink place.
- The call WoflanInfo\((h, \text{INFO\_SNC\_N\_NAME}, i)\) returns the name of the \(i\)-th node that is not strongly connected to a source place or a sink place.

### A.3 S-components

These functions all use the short-circuited WF-net.

- The call WoflanInfo\((h, \text{INFO\_NOF\_SCOM})\) returns the number of S-components.
- The call WoflanInfo\((h, \text{INFO\_NOF\_NOT\_SCOM})\) returns the number of nodes that are not covered by the S-components.
- The call WoflanInfo\((h, \text{INFO\_SCOM\_NOF\_N}, i)\) returns the number of nodes in the \(i\)-th S-component.
- The call WoflanInfo\((h, \text{INFO\_SCOM\_N\_NAME}, i, j)\) returns the name of the \(j\)-th node in the \(i\)-th S-component.
- The call WoflanInfo\((h, \text{INFO\_NOT\_SCOM\_N\_NAME}, i)\) returns the name of the \(i\)-th node that is not covered by the S-components.

### A.4 Place Invariants

These functions all use the short-circuited WF-net.

- The call WoflanInfo\((h, \text{INFO\_NOF\_PINV})\) returns the number of safe place invariants, that is, base place invariants containing only the weight 1 as non-zero weight.
The call WoflanInfo($h$, INFO_NOF_NOT_PINV) returns the number of places that are not covered by the safe place invariants.

The call WoflanInfo($h$, INFO_PINV_NOF_P, $i$) returns the number of places in the $i$-th safe place invariants.

The call WoflanInfo($h$, INFO_PINV_P_NAME, $i$, $j$) returns the name of the $j$-th place in the $i$-th safe place invariants.

The call WoflanInfo($h$, INFO_NOT_PINV_P_NAME, $i$) returns the name of the $i$-th place that is not covered by the safe place invariants.

The call WoflanInfo($h$, INFO_NOF_SPIN) returns the number of base place invariants.

The call WoflanInfo($h$, INFO_NOF_NOT_SPIN) returns the number of places that are not covered by the base place invariants.

The call WoflanInfo($h$, INFO_SPIN_NOF_P, $i$) returns the number of places in the $i$-th base place invariants.

The call WoflanInfo($h$, INFO_SPIN_P_NAME, $i$, $j$) returns the name of the $j$-th place in the $i$-th base place invariants.

The call WoflanInfo($h$, INFO_SPIN_P_NAME, $i$, $j$) returns the name of the $j$-th place in the $i$-th base place invariants.

The call WoflanInfo($h$, INFO_NOT_SPIN_P_NAME, $i$) returns the name of the $i$-th place that is not covered by the base place invariants.

The call WoflanInfo($h$, INFO_NOF_TINV) returns the number of base transition invariants.

The call WoflanInfo($h$, INFO_NOF_NOT_TINV) returns the number of transition that are not covered by the base transition invariants.

The call WoflanInfo($h$, INFO_TINV_NOF_T, $i$) returns the number of transitions in the $i$-th base transition invariants.

The call WoflanInfo($h$, INFO_TINV_T_NAME, $i$, $j$) returns the name of the $j$-th transition in the $i$-th base transition invariants.

The call WoflanInfo($h$, INFO_NOT_TINV_T_NAME, $i$) returns the name of the $i$-th transition that is not covered by the base transition invariants.

### A.5 Transition invariants

These functions all use the short-circuited WF-net.

The call WoflanInfo($h$, INFO_NOF_TINV) returns the number of base transition invariants.

The call WoflanInfo($h$, INFO_NOF_NOT_TINV) returns the number of transition that are not covered by the base transition invariants.

The call WoflanInfo($h$, INFO_TINV_NOF_T, $i$) returns the number of transitions in the $i$-th base transition invariants.

The call WoflanInfo($h$, INFO_TINV_T_NAME, $i$, $j$) returns the name of the $j$-th transition in the $i$-th base transition invariants.

The call WoflanInfo($h$, INFO_NOT_TINV_T_NAME, $i$) returns the name of the $i$-th transition that is not covered by the base transition invariants.
A.6 Free choice

- The call WoflanInfo(h, INFO_NOF_NFCC) returns the number of non-free-choice clusters.
- The call WoflanInfo(h, INFO_NFCC_NOF_N, i) returns the number of nodes in the i-th non-free-choice cluster.
- The call WoflanInfo(h, INFO_NFCC_N_NAME, i, j) returns the name of the j-th node in the i-th non-free-choice cluster.

A.7 Handles

These functions all use the short-circuited WF-net.

- The call WoflanInfo(h, INFO_NOF_TPH) returns the number of TP-handles.
- The call WoflanInfo(h, INFO_TPH_NOF_N1, i) returns the number of nodes on the first path of the i-th TP-handle.
- The call WoflanInfo(h, INFO_TPH_NOF_N2, i) returns the number of nodes on the second path of the i-th TP-handle.
- The call WoflanInfo(h, INFO_TPH_NOF_N1, i, j) returns the name of the j-th node on the first path of the i-th TP-handle.
- The call WoflanInfo(h, INFO_TPH_NOF_N2, i, j) returns the name of the j-th node on the second path of the i-th TP-handle.

A.8 Unboundedness

These functions all use the short-circuited WF-net. The functions related to unbounded places use the minimal coverability graph, while the functions related to unbounded sequences use a coverability graph.
The call WoflanInfo($h$, INFO_NOF_UNB_P) returns the number of unbounded places.

- The call WoflanInfo($h$, INFO_UNB_P_NAME, $i$) returns the name of the $i$-th unbounded place.
- The call WoflanInfo($h$, INFO_NOF_UNB_S) returns the number of unbounded sequences.
- The call WoflanInfo($h$, INFO_UNB_S_NOF_T, $i$) returns the number of transitions in the $i$-th unbounded sequence.
- The call WoflanInfo($h$, INFO_UNB_S_T_NAME, $i$, $j$) returns the name of the $j$-th transition in the $i$-th unbounded sequence.

A.9 Liveness

These functions all use the short-circuited WF-net.

- The call WoflanInfo($h$, INFO_NOF_DEAD_T) returns the number of dead transitions.
- The call WoflanInfo($h$, INFO_DEAD_T_NAME, $i$) returns the name of the $i$-th dead transition.
- The call WoflanInfo($h$, INFO_NOF_NLIVE_T) returns the number of non-live transitions.
- The call WoflanInfo($h$, INFO_NLIVE_T_NAME, $i$) returns the name of the $i$-th non-live transition.
- The call WoflanInfo($h$, INFO_NOF_NLIVE_S) returns the number of non-live sequences.
- The call WoflanInfo($h$, INFO_NLIVE_S_NOF_T, $i$) returns the number of transitions in the $i$-th non-live sequence.
- The call WoflanInfo($h$, INFO_NLIVE_S_T_NAME, $i$, $j$) returns the name of the $j$-th transition in the $i$-th non-live sequence.

A.10 Inheritance

- The call WoflanInfo($h$, INFO_INH_PT, $i$) returns 1 if the net with handle $i$ is a subclass under protocol inheritance [BA01] of the net with handle $h$, and 0 otherwise.
- The call WoflanInfo($h$, INFO_INH_PJ, $i$) returns 1 if the net with handle $i$ is a subclass under projection inheritance [BA01] of the net with handle $h$, and 0 otherwise.
The call WoflanInfo\((h, \text{INFO\_INH\_PP}, i)\) returns 1 if the net with handle \(i\) is a subclass under protocol/projection inheritance [BA01] of the net with handle \(h\), and 0 otherwise.

The call WoflanInfo\((h, \text{INFO\_INH\_LC}, i)\) returns 1 if the net with handle \(i\) is a subclass under life-cycle inheritance of the net with handle \(h\), and 0 otherwise.

A.11 Minimal coverability graph

These functions all use the short-circuited WF-net.

- The call WoflanInfo\((h, \text{INFO\_MCG\_CUTS})\) returns 1 if the minimal coverability graph of the net was cut [Fin93], and 0 otherwise. If it was cut, then substates exist in the reachability graph and the net cannot be sound.

A.12 Miscellaneous

- The call WoflanInfo\((h, \text{INFO\_DEADLINE})\) returns 1 if the deadline for generating a coverability graph was met, and 0 otherwise. If the deadline was not met, the coverability graph will be truncated to the starting node. To understand this, one should realize that generating a coverability graph can take considerable time. In certain situations, for instance, for the Web service application, this time might not be available. In such a situation, a deadline is set and the generation algorithm stops as soon as this deadline passes. Using this call, one can check whether the generation algorithm was able to meet the deadline or not.

- The call WoflanInfo\((h, \text{INFO\_REDUCE})\) reduces the net according to the liveness and boundedness preserving reduction rules (see Figure 4.11 on page 93) and returns the NULL-pointer.
APPENDIX B

The TPN format

This appendix describes the TPN format, which is one of Woflan’s native format. First, we give the extended BNF for the format, second, we give an example.

**B.1 Extended BNF**

1. Net = { Node }

Thus, a net contains any number of nodes.

2. Node = Place
   | Trans [ Label ] [ Ins ] [ Outs ]

Thus, a node is either a place or a transition. If the node is a transition, then it can optionally be followed by a label, inputs, and outputs (in that order).

3. Place = ‘place’ Name [ ‘init’ Number ]

Thus, a place contains the keyword ‘place’ and a name, optionally followed by the ‘init’ keyword and a number, which indicates the initial marking of this place.

4. Trans = ‘trans’ [ Name ]
5. Label = ‘~’ Name
6. Ins = ‘in’ { Arc }
7. Outs = ‘out’ { Arc }
8. Arc = Name [ Bound [ Bound | ‘inf’ ] ]
9. Name = Letter { Letter | Digit | ‘@’ | ‘.’ | '_' }

10. | ‘”’ { any character except ‘”’ } ‘”’
The TPN format

12 Number = Digit { Digit } 
13 Bound = Digit { Digit } ' ' { Digit } 
                [ 'e' [ '-' | '+' ] Digit { Digit } ] 
14 Letter = 'a' | ... | 'z' | 'A' | ... | 'Z' 
15 Digit = '0' | ... | '9' 

Every other character and everything from ‘--’ (two dashes) to a newline is treated as white space.

Note that we do not use bounds in this thesis: They were added to the TPN format for historical reasons. For sake of completeness, they are mentioned in the extended BNF.

B.2 Example

As an example for the TPN format, we have included the content of the TPN file that results after Staffware’s implementation process has been converted. Note that many transitions are labeled “ t” (a space followed by the character “t”): for Woflan, this label is the silent label τ.

```plaintext
1                      ---------------------------------------
2                      -- Generated by xfr2tpn, version 0.1 --
3                      --      (C) 2003, Eric Verbeek      --
4                      ---------------------------------------
5                      -- Using D:\Papers\PhD\Staffware\Revised.xfr as input file
6                      -- Using D:\Papers\PhD\Staffware\Revised.tpn as output file
7                      
8                      place "p1" init 1;
9                      trans "Connect/t1"~"Connect"
10                     in "Connect/p1"
11                     out "Merge/p1"
12                      ;
13                      trans "Order/t1"~"Order"
14                     in "Disconnect/p1" "Order/p1"
15                     out "orders=1 (4)/p1" "p3"
16                      ;
17                      trans "orders=1 (4)/t1"~" t"
18                     in "orders=1 (4)/p1"
19                     out "Confirm/p1"
20                      ;
21                      trans "orders=1 (4)/t2"~" t"
22                     in "orders=1 (4)/p1"
23                     out "Merge/p1"
24                      ;
25                      trans "Ship/t1"~"Ship"
```

286
Example

26     in "Ship/p1"
27     out "wait 9/p2"
28     ;
29 trans "Disconnect/t1"~"Disconnect"
30     in "Order/p1" "Disconnect/p1"
31     out "Reconnect/p1" "orders=0 (12)/p1"
32     ;
33 trans "orders=0 (12)/t1"~" t"
34     in "orders=0 (12)/p1"
35     out "Archive/p1"
36     ;
37 trans "orders=0 (12)/t2"~" t"
38     in "orders=0 (12)/p1"
39     out "Dummy/p1"
40     ;
41 trans "Merge/t1"~" Archive"
42     in "Reconnect/p1" "Archive/p1"
43     out "p3" "p3"
44     ;
45 trans "Reconnect/t1"~" Reconnect"
46     in "Archive/p1" "Reconnect/p1"
47     out "Merge/p1" "p3"
48     ;
49 trans "Archive/t1"~" Archive"
50     in "Reconnect/p1" "Archive/p1"
51     out "p3" "p3"
52     ;
53 trans "Confirm/t1"~" Confirm"
54     in "Confirm/p1" "p3"
55     out "Ship/p1" "Merge/p1"
56     ;
57 trans "Dummy/t1"~" t"
58     in "Dummy/p1"
59     out "wait 9/p1"
60     ;
61 trans "wait 9/t1"~" t"
62     in "wait 9/p2" "wait 9/p1"
63     out "Archive/p1" "p3"
64     ;
65 trans "t3"~" t"
66     in "p1"
67     out "Connect/p1" "p3" "p3" "p3" "p3" "p3" "p3" "p3"
68     ;
69 trans "t1"~" t"
70     in "p3" "p3" "p3" "p3" "p3" "p3" "p3" "p3"
71     out "p2"
72     ;
The TPN format

73  -- End of file
This appendix gives the informal description that was used for the case study of Section 4.6.

Some time ago the board of Somewhere University (SU) decided to open a travel agency at the campus. The new agency is supposed to organize both business and private trips for employees of SU. However, the service is not as the board expected. The most important complaint is that both the organization of a trip and the financial settlement take too long. Therefore, the board has started an investigation. Interviews with several people involved have provided the following process description. (To avoid confusion between employees of SU that want to book a trip and employees that are involved in the organization of the trip, in the remainder, the former are called clients.)

The whole process starts when someone drops in at the travel agency to book a trip. An employee of the agency registers all the relevant information of the client. The agency maintains a separate file for each trip. An important issue is whether the client wants to book a private trip, a business trip, or a combination of both. Approximately 20 percent of all the trips organized by the agency is private.

Private trips are easy. The agency has one employee dedicated to the organization of private trips. As soon as the wishes of a client are registered, she can start with the organization of the trip.

Business trips are more complicated. The agency has two employees for the organization of business trips (although one of them works only three days a week). For each trip, there is always a single employee responsible, who also carries out as many tasks as possible for this trip.
In this way, the service to clients should be guaranteed. For business trips, a client needs a travel permit. Usually, clients that are familiar with the process have already filled out a permit. Clients that arrive without a permit are given a blank permit that they can fill out later, after which they must return the permit to the agency. Travel permits are always checked before any other action is taken. If a permit is not filled out properly, it is returned to the client with the request to provide the missing information and send the permit back as soon as possible. In case a permit is not returned in time, the travel agency can no longer guarantee a timely organization of the trip. In the rare occasion that this happens, a notification is sent to the client and the file is closed. If a travel permit is okay, it is filed and the actual organization of the trip can start. First, however, a copy of the file is sent to the finance department of SU, because this department is responsible for the financial aspects of the trip.

An employee of the finance department of SU checks whether the client is allowed to make business trips paid by SU. The results of this check are sent to the travel agency in an internal memo. If the result is negative for the client, which is hardly ever the case because clients usually know when they are permitted to make business trips, the finance department does not make any payments. If the result is positive, the finance department makes an advance payment on the bank account of the client. It also pays any registration fees that might need to be paid in case of conference visits. Finally, it pays those flights of the trip that are made for business purposes. However, these payments can only be made after the finance department has received detailed pricing information from the travel agency. After all the necessary payments have been made, the finance department is no longer involved in the preparations of the trip. However, after the client returns, the finance department handles the client’s declaration (see below).

To prepare a trip (private or business), the travel agency always starts with flight arrangements. If a trip involves one or more flights, the responsible employee of the travel agency starts by preparing a flight schedule that includes departure and arrival times of all flights as well as pricing information. Then, the client is called to approve the schedule. If the client does not approve the schedule, a new proposal is prepared and the client is contacted again. When a client approves the schedule, arrangements must be made to pay the flight(s). In case the trip is private, an appointment is made with the client to pay cash or by credit card. In case the trip is (partly) business, the travel agency has to wait for the memo of the finance department which states whether or not the client is allowed to make business trips for SU. If the memo is negative, the employee of the travel agency responsible for the trip calls the client to explain the problem. If the client still wants to make the trip, he or she has to pay all the costs and an appointment is made to
pay for the flights. However, often the client decides to cancel the trip, in which case the file is closed. If the memo is positive, the travel agency determines the costs of business flights and, if applicable, the costs of private flights. Relevant information on business flights is sent to the finance department that handles the actual payment. In case of private flights, the client is contacted to make an appointment to arrange the payment.

The internal memo that the travel agency receives from the finance department, is also used to determine whether a request must be sent to the in-house bank office (which is situated at the campus close to the travel agency) to prepare cash and travel cheques for the client. Such a request is always made when a business trip is allowed. (In case of private trips, the client has to take care of acquiring cash and cheques him- or herself.)

The task of the bank in the process is very straightforward. Upon receipt of a request, a bank employee prepares cash and travel cheques and sends them to the travel agency. If a client returns cash and/or cheques after the trip, information about the exact amount that is used by the client is sent to the finance department. The finance department needs this information to process the client’s declaration. In case a client does not return cash or cheques in time, the amount supposedly spent by the client is fixed to the value of the cash and cheques handed out to the client before the trip.

After flight arrangements have been made and any private flights have been paid, the responsible employee of the travel agency books hotels and makes reservations for local transportation (train, car, etc.). She also prints vouchers for any hotels that are booked. When cash and cheques have been received from the bank and all flight tickets have been received from the central office of the travel agency in SomewhereElse where they are printed, the employee puts all the documents together in a handy folder for the client. The agency has to make sure that everything is ready at least three working days before the trip starts, because, then, the client picks up the documents. At that point, the involvement of the agency with the trip is finished. In case of a private trip, this also means that the process is complete. In case of a business trip, however, the declaration of the client still needs to be processed.

As mentioned, the finance department takes care of processing declarations. When it has received a client’s declaration and the necessary information of the bank, an employee of the finance department processes the declaration and calculates the balance. The result must be approved by the director of the finance department. In case of mistakes, the employee must make the necessary corrections. After the
declaration has been approved by the director, the balance is settled with the next salary payment of the client. In addition, the total cost of the trip is deducted from the travel budget of the faculty or other unit where the client is employed. If a client does not hand in his or her declaration in time (within a month after completion of the trip), the finance department assumes that the total cost of the trip equals the sum of the advance payment and the value of the cash and cheques given to the client.

The board of SU thinks that the main reason why the above process takes so long is that the co-ordination between the three departments involved is poor. It believes that a workflow system might provide a solution. As a starting point, it would like to receive a report covering the following subjects.

1. A resource classification of all the resources involved in the current process, distinguishing roles and groups.
2. A process model of the current situation developed in Protos, including information about roles and triggers.
3. An analysis of the resource classification and the process model, using the guidelines for process (re-)design discussed in the book and the slides.
4. An improved resource classification/process model developed in Protos, based on the results of the analysis. (Include only the graphical representation of the resource classification/process model and any information which is new compared to the original resource classification/process model.)
This appendix gives the Data Type Definition (DTD) [BPSMM00] of the eXchangeable Routing Language (XRL) [AK02, AVK01a, AVK01b, VA04]. For the XML Schema Definition (XSD), we refer to the URL http://is.tm.tue.nl/staff/everbeek/download/xrl 1.0.xsd.

```xml
<!ENTITY % routing_element
  "task|sequence|any_sequence|choice
  |condition|parallel_sync|parallel_no_sync
  |parallel_part_sync|parallel_part_sync_cancel
  |wait_all|wait_any|while_do|terminate"
>
<!ELEMENT route
  (%routing_element;), event*)
>
<!ATTLIST route
  name ID #REQUIRED
  created_by CDATA #IMPLIED
  date CDATA #IMPLIED
>
<!ELEMENT task
  (event*)
>
<!ATTLIST task
  name ID #REQUIRED
  address CDATA #REQUIRED
  role CDATA #IMPLIED
  doc_read NMTOKENS #IMPLIED
  doc_update NMTOKENS #IMPLIED
  doc_create NMTOKENS #IMPLIED
  result CDATA #IMPLIED
  status (ready|running|enabled
```
<!ELEMENT event EMPTY>

<!ATTLIST event
    name ID #REQUIRED>

<!ELEMENT sequence
    (%routing_element;|state)+>

<!ELEMENT any_sequence
    (%routing_element;)+>

<!ELEMENT choice
    (%routing_element;)+>

<!ELEMENT condition
    (%true|false)+>

<!ATTLIST condition
    condition CDATA #REQUIRED>

<!ELEMENT true
    (%routing_element;)
>
<!ELEMENT false
    (%routing_element;)
>
<!ELEMENT parallel_sync
    (%routing_element;)+>

<!ELEMENT parallel_no_sync
    (%routing_element;)+>

<!ELEMENT parallel_part_sync
    (%routing_element;)+>

<!ATTLIST parallel_part_sync
    number NMTOKEN #REQUIRED>

<!ELEMENT parallel_part_sync_cancel
    (%routing_element;)+>

<!ATTLIST parallel_part_sync_cancel
    number NMTOKEN #REQUIRED>
<!ELEMENT wait_all
  ((event_ref|timeout)+)
>
<!ELEMENT wait_any
  ((event_ref|timeout)+)
>
<!ELEMENT event_ref
  EMPTY
>
<!ATTLIST event_ref
  name IDREF #REQUIRED
>
<!ELEMENT timeout
  (%routing_element;)?
>
<!ATTLIST timeout
  time CDATA #REQUIRED
type (relative|s_relative|absolute)
  "absolute"
>
<!ELEMENT while_do
  (%routing_element;)
>
<!ATTLIST while_do
  condition CDATA #REQUIRED
>
<!ELEMENT terminate
  EMPTY
>
<!ELEMENT state
  EMPTY
>

295


299

Bibliography


nets. In G. De Michielis and M. Diaz, editors, Application and Theory of Petri Nets 1995, Proceed-
ings, volume 935 of Lecture Notes in Computer Science, pages 25–44, Torino, Italy, June

[BCH+03] J. Billington, S. Christensen, K.M. van Hee, E. Kindler, O. Kummer, L. Petrucci, R.D.J. Post,
Aalst and Best [AB03], pages 483–507.


mCRL: A toolset for analysing algebraic specifications. In G. Berry, H. Comon, and A. Finkel,
editors, Proceedings of the 13th International Conference on Computer Aided Verification
(CAV’01), volume 2102 of Lecture Notes in Computer Science, pages 250–254. Springer, Ber-

[BPSMM00] T. Bray, J. Paoli, C.M. Sperberg-McQueen, and E. Maler. eXtensible Markup Language

son-Wesley, Reading, Massachusetts, USA, 1998.

[BV95] J.C.M. Baeten and C. Verhoef. Concrete process algebra. In S. Abramsky, Dov M. Gabbay,
and T.S.E. Maibaum, editors, Handbook of Logic in Computer Science, volume 4, Semantic


[Cas98] R. Casonato. Gartner Group research note 00057684, production-class workflow: A view of

[CCPP98] F. Casati, S. Ceri, B. Pernici, and G. Pozzi. Workflow evolution. Data and Knowledge Engi-


[CKM01] S. Christensen, L.M. Kristensen, and T. Mailund. A sweep-line method for state space explo-
ration. In Proceedings of TACAS’01, volume 2031 of Lecture Notes in Computer Science, pages

of algorithms for computation of minimal P-semiflows. In G. Rozenberg, editor, Advances in
Petri Nets 1990, volume 483 of Lecture Notes in Computer Science, pages 79–112. Springer,
Berlin, Germany, 1990.


Sound Workflow Specification. PhD thesis, Technische Universität Berlin, Berlin, Germany,
August 2003.
<table>
<thead>
<tr>
<th>Reference</th>
<th>Title</th>
</tr>
</thead>
</table>


[Hac72] M. Hack. Analysis of production schemata by Petri nets. Master’s thesis, Department of Electrical Engineering, Massachusetts Institute of Technology, Cambridge, Massachusetts, USA, February 1972. We were unable to get a copy of this masters thesis.


INDEX

Symbols

&CONDITION&  206
  see also COSA transfer
.scr file, see COSA script file
.tpn file, see TPN file format
.xfr file, see Staffware procedure file

Numerics
1-soundness  33

A
  abstraction  70
  see also hiding
  abstraction from control data  21
  abstraction from data  21
  abstraction from production data  21
  action  39, 67
  action, silent, see internal action
  action, unobservable, see internal action
  activity  15
  activity, COSA, see COSA activity
  activity, MQSeries, see MQSeries activity
  activity, Protos, see Protos activity
  algorithm, backtracking, see backtracking algorithm
  algorithm, elementary transition invariants, see elementary transition invariants algorithm
  algorithm, exhaustive search, see exhaustive search algorithm
  AND-join  15, 18, 101, 102, 183, 194, 198, 203, 210, 213, 246
  see also mismatch, OR-join, XOR-join
  AND-OR mismatch  225
  AND-split  15, 18, 102, 194, 210, 213, 239, 246
  see also mismatch, OR-split, XOR-split
  any_sequence, see XRL any_sequence
  arc  38, 39
  arc weight  40
  arc weight, removing  161
  arc, bidirectional  38
  architecture, Woflan  217, 232
  assumption  74
  see also presupposition
  bag  42
  see also marking
  bag, extended, see extended bag
  bag, transition, see transition bag
  begin place  46
  behavioral error sequence  33, 84, 272
  see also non-live sequence, unbounded sequence
  bisimilarity, branching, see branching bisimilarity
  block constraint  119, 128
  block set  119, 124
  blocking  8, 25, 69, 117, 125, 219
  see also encapsulation
  bounded place  60, 225
  see also k-bounded place, safe place
  bounded system  60, 77, 88
  see also k-bounded system, safe system
  boundedness  33, 66, 78, 92, 161, 271
| BPEL4WS, see Business Process Execution Language for Web Services (BPEL4WS) |  |
| BPML, see Business Process Modeling Language (BPML) |  |
| branch, see team |  |
| branching bisimilar systems 68 |  |
| branching bisimilarity 68, 117, 125, 218, 255 |  |
| branching bisimulation 68, 121 |  |
| business process 14 |  |
| workflow management 13 |  |
| Business Process Execution Language for Web Services (BPEL4WS) 142 |  |
| Business Process Modeling Language (BPML) 142 |  |
| business process, change in, see change in business process |  |

| C |  |
| capacity, place, see place capacity |  |
| case 14 |  |
| task, see work item |  |
| case-based 14 |  |
| change in business process 1 |  |
| error 1 |  |
| workflow management system 1 |  |
| choice, see XRL choice |  |
| cluster, non-free-choice, see non-free-choice cluster |  |
| complete, option to, see option to complete |  |
| completion, proper, see proper completion |  |
| complex definition, see Woflan:complex definition |  |
| complex router, see Staffware complex router |  |
| complexity 34 |  |
| computing-science student 111 |  |
| condition 18, 224, 225 |  |
| task 14 |  |
| condition, exit, see MQSeries exit condition |  |
| condition, improper, see improper condition |  |
| condition, join, see MQSeries join condition |  |
| condition, proper, see proper condition |  |
| condition, Staffware, see Staffware condition |  |
| condition, start, see start condition |  |
| condition, useless, see useless condition |  |
| condition, XRL, see XRL condition |  |
| confusion 101, 225, 226, 239 |  |
| see also free-choice cluster |  |
| connected net 49 |  |
| connected net, strongly, see strongly connected net |  |
| connection, Protos, see Protos connection |  |
| connector, MQSeries, see MQSeries connector |  |
| consistent XML document 144 |  |
| constraint 118, 124, 251, 275 |  |
| constraint, block, see block constraint |  |
| constraint, hide, see hide constraint |  |
| control data 18 |  |
| control data, abstraction from, see abstraction from control data |  |
| control-flow |  |
| workflow process definition (WPD) 18 |  |
| control-flow perspective 18, 19, 98, 148 |  |
| COSA activity 206 |  |
| AND/OR/XOR 210 |  |
| COSA end activity 206 |  |
| COSA mapping 208, 213, 223 |  |
| COSA Network Editor (CONE) 2 |  |
| COSA process definition 206 |  |
| soundness 211 |  |
| COSA process definition, Original 207 |  |
| COSA process definition, Revised 207 |  |
| COSA requirement 206 |  |
| COSA requisition process 2 |  |
| COSA script file 223 |  |
| COSA start activity 206 |  |
| COSA subprocess 206, 223 |  |
| COSA transfer 206 |  |
| condition 206, 209, 212 |  |
| see also &CONDITION& |  |
| COSA Workflow 29, 181, 206 |  |
| coverability graph 62, 89, 92, 275 |  |
| see also minimal coverability graph, reachability graph |  |
| partitioning for unbounded sequences 90 |  |
| coverability graph, minimal, see minimal coverability graph |  |
| coverability graph, restricted, see restricted coverability graph |  |
| coverability set 29, 181, 206 |  |
| cycle 125, 126 |  |
| cycle, elementary, see elementary cycle |  |
| cycle, observable-label support of, see observable-label support |  |
| cycle |  |

| D |  |
| data 15 |  |
| data element 196, 269 |  |
| data perspective 18, 21, 268, 269 |  |
| Data Type Definition (DTD) 144 |  |
| data, control, see control data |  |
| data, production, see production data |  |
| dead task 107, 225, 228 |  |
| see also dead transition |  |
| dead task, no, see no dead task |  |
| dead transition 58, 82, 83, 107, 225 |  |
| see also dead task |  |
| deadlock 1 |  |
| see also error |  |
| change in business process |  |
| resource 20 |  |
| department, see team |  |
Index

diagnosis process 33, 96, 221, 224, 226, 237, 241, 272
milestone, see milestone
diagnostic information, see Woflan:diagnostic information
directed path 49
DTD, see Data Type Definition (DTD)
due-date, see task perspective
duration, see task perspective

diagnosis process 33, 96, 221, 224, 226, 237, 241, 272
milestone, see milestone
diagnostic information, see Woflan:diagnostic information
directed path 49
DTD, see Data Type Definition (DTD)
due-date, see task perspective
duration, see task perspective

E
ebXML, see Electronic Business using XML (ebXML)
E-commerce 142
Eindhoven 111
Electronic Business using eXtensible Markup Language (ebXML) 142
elementary cycle 130
elementary extended non-self controlling net 52
elementary path 50
elementary transition invariant 131
elementary transition invariants algorithm 133, 234
enabled transition 44
encapsulation 69
see also blocking
deep place 46
EPC, see Event-driven Process Chain (EPC)
error change in business process 1
deadlock, see deadlock
deadlock 186
livelock, see livelock
S-component 75
error, behavioral sequence, see behavioral error sequence
event handler, see XRL event handler
event, see XRL event
Event-driven Process Chain (EPC) 29, 32
eXchangeable Routing Language (XRL) 29, 33, 141
exhaustive search algorithm 117, 121, 133, 192, 251, 261, 266
see also backtracking algorithm, life-cycle inheritance
exit condition, see MQSeries exit condition
extended bag 62
extended enterprise 142
extended free-choice net 52
extended marking 62
extending label 117, 121, 125
extending task 8
eXtensible Stylesheet Language Transformations (XSLT) 144, 233

F
false token 186
file format, TPN, see TPN file format
finite marking 62
firing rule 44, 67
firing sequence 57
first trade problem 143
flexible workflow management system 23, 30
flow diagram, see workflow process definition (WPD)
FlowMake 30
free-choice cluster 101
free-choice net, extended, see extended free-choice net

G
generalized soundness 272
Graphical Workflow Definer, see Staffware
Graphical Workflow Definer green marking 86, 90

H
handle 50
see also mismatch
handle, PT, see PT-handle
handle, TP, see TP-handle
hide constraint 119, 128
hide set 119, 124
hiding 9, 25, 70, 117, 125, 219
see also abstraction

I
IBM MQSeries Workflow, see MQSeries
identifier 39
improper condition 105
see also unbounded place
improper scenario 106, 225, 249
see also unbounded sequence
industrial-engineering student 111
infinite marking 62, 90
inheritance 8, 23
Unified Modeling Language (UML) 25
inheritance, life-cycle, see life-cycle inheritance
inheritance, projection, see projection inheritance
inheritance, protocol, see protocol inheritance
inheritance, protocol/projection, see protocol/projection inheritance
initial state 40, 43
input node 40
internal action 39
Internet 142
inter-organizational workflow process 22, 142, 266
error 23
invariant, \( P \), see place invariant
invariant, place, see place invariant
invariant, \( S \), see place invariant
invariant, semi-positive, see semi-positive invariant
invariant, \( T \), see transition invariant
invariant, transition, see transition invariant

\[ \text{Index} \]

\section*{J}
join condition, see MQSeries join condition

\section*{K}
Karlsruhe 111
Karp-Miller graph, see coverability graph
k-bounded place 60
\( \text{see also} \) bounded place, safe place
k-bounded system 60
\( \text{see also} \) bounded system, safe system
k-safe place, see k-bounded place
k-safe system, see k-bounded system
k-soundness 272

\section*{L}
L0-live transition, see dead transition
L4-live system, see live system
L4-live transition, see live transition
label 39
label, blocked, see blocked label
label, extending, see extending label
label, hidden, see hidden label
label, observable, see observable label
label, silent, see silent label
labeled P/T net, see net
labeled reachability graph 67
life-cycle inheritance 9, 23, 25, 34, 70, 117, 125, 217, 230, 251, 255, 256, 257
\( \text{see also} \) backtracking algorithm, exhaustive search algorithm
MQSeries process definition 192
workflow net (WF-net) 71
link, see Staffware link
live system 58, 84
live task 225
live transition 58, 225
livelock 1
\( \text{see also} \) non-live sequence

\section*{M}
management view
\( \text{see also} \) inheritance
workflow process definition (WPD) 6
mapping 26, 98
COSA Workflow 208, 213, 223, 269
IBM MQSeries Workflow 185, 235, 269
Protos 213, 246, 269
Staffware 196, 239, 269
XRL 154
markable place 122
marked net, see system
marking 42
bag 42
state 42
marking, extended, see extended marking
marking, finite, see finite marking
marking, green, see green marking
marking, infinite, see infinite marking
marking, reachable, see reachable marking
marking, red, see red marking
marking, yellow, see yellow marking
meta-model 16
milestone 98
minimal coverability graph 64, 275
\( \text{see also} \) coverability graph, reachability graph
minimal coverability set 64
\( \text{see also} \) coverability set
minimal observable-label support 129
minimality 129
mismatch 226, 239
\( \text{see also} \) AND-join, AND-split, handle, XOR-join, XOR-split
model checking 31, 276
Unified Modeling Language (UML) activity diagram 31
MQSeries 29, 181, 183
FlowMake, see FlowMake
MQSeries activity 183, 186
block 183
process 183
program 183
MQSeries connector 183, 188
control 183
default 183, 191
MQSeries end activity 183
MQSeries exit condition 183
MQSeries join condition 183
\( \text{see also} \) AND-join, OR-join
MQSeries mapping 34, 185, 235, 269
MQSeries process definition 183, 184
life-cycle inheritance 192
no dead task 190
option to complete 190
proper completion 190
soundness 190
MQSeries process definition, Original, see Original MQSeries process definition
MQSeries process definition, Revised, see Revised MQSeries process definition
MQSeries start activity 183
MQSeries subprocess 190
muCRL (mCRL) 269, 276
multiset, see bag

net 37, 225
arc, see arc
boundedness, see boundedness
firing rule, see firing rule
liveness, see liveness
marking, see marking
node, see node
place, see place
state machine, see state machine
state, see state
subnet, see subnet
transition, see transition
workflow (WF-net), see workflow net (WF-net)
net, connected, see connected net
net, elementary extended non-self controlling, see
elementary extended non-self controlling net
net, free-choice, see free-choice net
net, marked, see system
net, Original, see Original net
net, partial subnet, see partial subnet
net, Revised, see Revised net
net, S-coverable, see S-coverability
net, strongly connected, see strongly connected net
net, well-handled, see well-handled net
net, well-structured, see well-structured net
Network Editor, see COSA Network Editor (CONE)
no dead task 8, 24, 107, 271
MQSeries process definition 190
Staffware process definition 202
node 38
path, see path
postset, see postset
preset, see preset
node, input, see input node
node, output, see output node
node, sink, see sink node
node, source, see source node
non-free-choice cluster 225
see also confusion
non-live scenario 225
non-live sequence 85, 88, 108
see also behavioral error sequence, locking
scenario, unbounded sequence
partitioning of reachability graph 86
non-live task 108
see also non-live transition
non-live transition 81, 83, 228
see also non-live task
normal link, see Staffware link:normal

object, see Staffware object
observable label 125
observable-label support 125
cycle 126
transition invariant 131
observable-label support of cycles 126
observable-label support, minimal, see minimal
observable-label support
occurrence graph, see reachability graph
occurrence sequence 57
operational perspective 19, 22
option to complete 8, 24, 81, 84, 228, 271
MQSeries process definition 190
Staffware process definition 202
OR-AND mismatch 225
organizational unit, see team
Original COSA process definition 207
Original MQSeries process definition 184
Original net 46
Original Protos process definition 213
Original Staffware process definition 195, 230
Original system 46
Original XRL process definition 148
OR-join 15, 18, 183, 210, 213
see also AND-join, XOR-join
OR-split 15, 18, 210, 213
see also AND-split, XOR-split
output node 40

P
P/T net, labeled, see net
parallel_no_sync, see XRL parallel_no_sync
parallel_sync, see XRL parallel_sync
partial subnet 41
path 49
handle, see handle
path, directed, see directed path
path, elementary, see elementary path
pattern 29, 141, 143
workflow process definition (WPD) 29
perspective, control-flow, see control-flow
perspective
perspective, data, see data perspective
perspective, operational, see operational perspective
perspective, resource, see resource perspective
perspective, task, see task perspective
Petri net 26, 37
application domain 37
Graph 39
Woflan 7
Index

Petri-Net Markup Language (PNML) 234
PIF, see Process Interchange Format (PIF)
P-invariant, see place invariant
place 38, 225
place capacity 40
place covered by place invariant 78
see also uniform invariant cover, weighted invariant cover
place invariant 56, 104, 168, 202, 225
place invariant, place covered by, see place covered by place invariant
place, begin, see begin place
place, bounded, see bounded place
place, end, see end place
place, k-bounded, see k-bounded place
place, k-safe, see k-bounded place
place, marked, see marked place
place, safe, see safe place
place, unmarkable, see unmarkable place
place/transition net (P/T net) 39
PNML, see Petri-Net Markup Language (PNML)
post(n) 40
post(N,n) 40
postset 40
pre(n) 40
pre(N,n) 40
presupposition 74
see also assumption
priority, see task perspective
procedure, see workflow process definition (WPD)
process
see also Staffware process
process activity, see Staffware process activity
process algebra 269
process definition 225
Process Interchange Format (PIF) 29
process, diagnosis, see diagnosis process
process, requisition, see requisition process
process, travel agency, see travel agency process
process, web portal, see web portal process
product independent, see Woflan:product independent
production data 18
production data, abstraction from, see abstraction from production data
program activity, see MQSeries activity:program projection inheritance 9, 23
proper completion 8, 24, 79, 88, 104, 271
MQSeries process definition 190
Staffware process definition 202
proper condition 225
proper workflow process definition 227
proper workflow process definition (WPD) 98
see also bounded system, short-circuited WF-net, workflow system (WF-system)
protocol inheritance 8, 23
protocol/projection inheritance 9, 23
Protos 111, 182, 212, 224, 243, 261
mapping 213, 224, 269
requisition process 6
Protos activity 213
Protos connection 213
Protos end activity 213
Protos mapping 246
Protos process definition 213
mapping 213
Protos process definition, Original, see Original Protos process definition
Protos start activity 213
Protos status 213
Protos subprocess 213, 243
PT-handle 50, 103, 225

R
reachability 58
reachability graph 59, 92
see also coverability graph, minimal coverability graph
partitioning for non-live sequences 86
reachability graph, labeled, see labeled reachability graph
reachability set 58
reachable marking 57
reactive workflow net (WF-net) 32
red marking 86, 90
reduction 92
boundedness 93, 177
free-choice 94
liveness 93, 177
restricted coverability graph, see restricted coverability graph
relaxed soundness 32, 272
requirement, see COSA requirement
requisition process 2, 276
COSA 2
Protos 6
Staffware 4
resource 15, 18
see also task perspective
abstraction from control data, see abstraction from control data
resource perspective 18
restricted coverability graph 95
see also coverability graph
Revised COSA process definition 207
Revised MQSeries process definition 184
Revised net 46
Revised Staffware process definition 196, 225, 233
Revised system 46
Revised XRL process definition 148
role 18
route element, see XRL route element
route, see XRL route
router, see Staffware router
routing construct 46, 193, 271
routing definition, see workflow process definition (WPD)
routing element, see XRL routing element

S
safe place 60
see also bounded place, k-bounded place
safe system 60, 77
see also bounded system, k-bounded system
safe workflow process definition 182
scenario, improper, see improper scenario
scenario, locking, see locking scenario
S-component 54, 75, 99, 225
see also thread of control
error 75
S-coverability 54, 74, 76, 77, 104
see also thread of control cover
S-coverable, see S-coverability
semi-positive invariant 56
sequence
see also XRL sequence
sequence, firing, see firing sequence
sequence, occurrence, see occurrence sequence
set, block, see block set
set, hide, see hide set
short-circuited WF-net 48, 225
short-circuited WF-system 66, 127
sink node 47
sink place 98, 200, 225
S-invariant, see place invariant
sound WF-net 65, 74, 217
sound workflow process definition (WPD) 23, 98
see also short-circuited WF-net, sound WF-net
soundness 8, 23, 24, 33, 65, 237, 271
MQSeries process definition 190
no dead task, see no dead task
option to complete, see option to complete
proper completion, see proper completion
Staffware process definition 202
WF-net, see sound WF-net
workflow process definition (WPD) 23
soundness, 1, see 1-soundness
soundness, generalized, see generalized soundness
soundness, k, see k-soundness
soundness, relaxed 32
soundness, relaxed, see relaxed soundness
source node 47
source place 98, 200, 225
spanning tree 107, 108
Staffware 181, 193
Staffware condition 194
Staffware link 193
normal 198
wait 198
withdraw 198
Staffware mapping 34, 196, 239, 269
Staffware object 193, 197
start 193
step 193, 204
stop 195
subprocess 194, 224
wait 194, 203, 239
Staffware procedure file 224, 225, 233
Staffware process definition 193
no dead task 202
option to complete 202
proper completion 202
semantics 203, 270
soundness 202
Staffware process definition, Original, see Original
Staffware process definition
Staffware process definition, Revised, see Revised
Staffware process definition
Staffware requisition process 4
Staffware router 194
Staffware subprocess 193, 197, 198
standard workflow process definition 182
start activity, COSA, see COSA start activity
start activity, MQSeries, see MQSeries start activity
start activity, Protos, see Protos start activity
start condition 225
start, see Staffware object:start
state
marking 42
token 42
state machine 53
state, initial, see initial state
status, see Protos status
step, see Staffware object:step
strongly connected net 49, 98
strongly unconnected node 225
structured workflow process definition 182
students, computing-science, see computing-science students
students, industrial-engineering, see industrial-engineering students
subnet 41, 101
S-component, see S-component
subnet, partial, see partial subnet
subprocess 193
COSA, see COSA subprocess
MQSeries, see MQSeries subprocess
Protos, see Protos subprocess
Staffware, see Staffware subprocess
substate 105
successful terminal state 85, 90, 121
support, observable-label, see observable-label support
synchronizing workflow process definition 182
system 43
   initial state 43
   successful terminal state 43
system, bounded, see bounded system
system, k-bounded, see k-bounded system
system, k-safe, see k-bounded system
system, live, see live system
system, Original, see Original system
system, Revised, see Revised system
system, safe, see safe system
systems, branching bisimilar, see branching bisimilar systems

T
task 14, 18, 193, 225
   see also XRL task
case, see work item
condition 14
order 14
work item, see work item
workflow process definition (WPD) 14
task perspective 18, 22
task, dead, see dead task
task, extending, see extending task
task, no dead, see no dead task
task, non-live, see non-live task
task, useless, see useless task
team 18
terminal state, successful, see successful terminal state
terminate handler, see XRL terminate handler
terminate, see XRL terminate
thread of control 99, 225, 246, 247
   see also S-component
thread of control cover 99, 100
   see also S-coverability
thread-of-control cover 226
T-invariant, see transition invariant
token 40, 42
   token, false, see false token
token, true, see true token 186
TP-handle 51, 103, 225
TPN file format 221, 223
transfer, see COSA transfer
transition 38, 224, 225
transition bag 130
transition invariant 56, 275
transition invariant, elementary, see elementary transition invariant
transition invariant, observable-label support of, see observable-label support of transition invariant
transition system
   see also reachability graph
transition, dead, see dead transition
transition, enabled, see enabled transition
transition, L0-live, see dead transition
transition, L4-live, see live transition
transition, live, see live transition
transition, non-live
   see also non-live task
transition, non-live, see non-live transition
transition system
travel agency process 110
trigger 15, 22
   see also task perspective
token 40, 42
   see also XRL token
true token 186

U
U3 9
UML, see Unified Modeling Language (UML)
unbounded place 79, 83, 96, 104, 105
   see also improper condition
unbounded sequence 89, 91, 106, 225
   see also behavioral error sequence, improper scenario, non-live sequence
partitioning of coverability graph 90
Unified Modeling Language (UML) 29
   activity diagram 31
   model checking 31, 276
class diagram 25, 144, 243
component architecture 217, 232
inheritance 25
uniform invariant 225
uniform invariant cover 104
   see also place covered by place invariant, weighted invariant cover
uniform-invariant cover 226
unmarkable place 122
useless condition 225
useless task 225, 244

V
verification
   workflow management system 2
   XRL process definition 167
visible label support of transition invariants 131

W
wait link, see Staffware link:wait
wait, see Staffware object:wait
wait_all, see XRL wait_all
wait_any, see XRL wait_any
web portal process 45
Web Service Choreography Interface (WSCI) 142
weight, arc, see arc weight
weighted invariant 225
<table>
<thead>
<tr>
<th>Entry</th>
<th>Page(s)</th>
<th>Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>weighted invariant cover</td>
<td>104</td>
<td>see also place covered by place invariant, uniform invariant cover</td>
</tr>
<tr>
<td>weighted-invariant cover</td>
<td>227</td>
<td></td>
</tr>
<tr>
<td>well-handled net</td>
<td>51</td>
<td></td>
</tr>
<tr>
<td>well-structured net</td>
<td>52, 76, 100, 103</td>
<td></td>
</tr>
<tr>
<td>WfMC, see Workflow Management Coalition (WfMC)</td>
<td>63</td>
<td></td>
</tr>
<tr>
<td>WF-net, see workflow net (WF-net)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>WF-system, see workflow system (WF-system)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>while_do, see XRL while_do</td>
<td></td>
<td></td>
</tr>
<tr>
<td>withdraw link, see Staffware link</td>
<td></td>
<td></td>
</tr>
<tr>
<td>withdraw</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Woflan 7, 217, 267</td>
<td></td>
<td></td>
</tr>
<tr>
<td>complex definition 7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>component architecture 217, 232</td>
<td></td>
<td></td>
</tr>
<tr>
<td>diagnostic information 7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Petri net 7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>product independent 7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Woflan tool 7, 33, 217, 267</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Web service</td>
<td>232</td>
<td></td>
</tr>
<tr>
<td>Windows</td>
<td>218, 224</td>
<td></td>
</tr>
<tr>
<td>WoflanClose 221</td>
<td></td>
<td></td>
</tr>
<tr>
<td>WoflanInfo 221</td>
<td></td>
<td></td>
</tr>
<tr>
<td>WoflanOpen 221</td>
<td></td>
<td></td>
</tr>
<tr>
<td>work item 15</td>
<td></td>
<td></td>
</tr>
<tr>
<td>activity, see activity</td>
<td></td>
<td></td>
</tr>
<tr>
<td>resource, see resource</td>
<td></td>
<td></td>
</tr>
<tr>
<td>trigger, see trigger</td>
<td></td>
<td></td>
</tr>
<tr>
<td>workflow instance, see case</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Workflow Management Coalition (WfMC) 14, 26, 224</td>
<td></td>
<td></td>
</tr>
<tr>
<td>building block 14</td>
<td></td>
<td></td>
</tr>
<tr>
<td>see also OR-join, OR-split, AND-join, AND-split</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Workflow Process Definition Language (WPDL), see Workflow Process Definition Language (WPDL)</td>
<td>26</td>
<td></td>
</tr>
<tr>
<td>XML Process Definition Language (XPDL), see XML Process Definition Language (XPDL)</td>
<td>26</td>
<td></td>
</tr>
<tr>
<td>workflow management system 13</td>
<td></td>
<td></td>
</tr>
<tr>
<td>change in business process 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>verification 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>workflow management system, expressive power of 268</td>
<td></td>
<td></td>
</tr>
<tr>
<td>workflow management system, flexible, see flexible workflow management system</td>
<td>26</td>
<td></td>
</tr>
<tr>
<td>workflow meta-model, see meta-model</td>
<td></td>
<td></td>
</tr>
<tr>
<td>workflow net (WF-net) 26, 37, 46</td>
<td></td>
<td></td>
</tr>
<tr>
<td>life-cycle inheritance 71, 217</td>
<td></td>
<td></td>
</tr>
<tr>
<td>soundness 65, 66, 217</td>
<td></td>
<td></td>
</tr>
<tr>
<td>workflow net (WF-net), reactive 32</td>
<td></td>
<td></td>
</tr>
<tr>
<td>workflow net (WF-net), short-circuited, see short-circuited WF-net</td>
<td>26</td>
<td></td>
</tr>
<tr>
<td>workflow pattern, see pattern</td>
<td></td>
<td></td>
</tr>
<tr>
<td>workflow process 14</td>
<td></td>
<td></td>
</tr>
<tr>
<td>workflow process definition 225</td>
<td></td>
<td></td>
</tr>
<tr>
<td>workflow process definition (WPD) 7, 14, 46, 98, 217, 225, 244</td>
<td></td>
<td></td>
</tr>
<tr>
<td>control-flow 18</td>
<td></td>
<td></td>
</tr>
<tr>
<td>management view 6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>minimal requirements, see soundness</td>
<td></td>
<td></td>
</tr>
<tr>
<td>pattern 29</td>
<td></td>
<td></td>
</tr>
<tr>
<td>task 14</td>
<td></td>
<td></td>
</tr>
<tr>
<td>workflow process definition (WPD), proper, see proper workflow process definition</td>
<td>217</td>
<td></td>
</tr>
<tr>
<td>workflow process definition (WPD), safe, see safe workflow process definition</td>
<td>217</td>
<td></td>
</tr>
<tr>
<td>workflow process definition (WPD), sound, see soundness</td>
<td>217</td>
<td></td>
</tr>
<tr>
<td>workflow process definition (WPD), standard, see standard workflow process definition</td>
<td>217</td>
<td></td>
</tr>
<tr>
<td>workflow process definition (WPD), structured, see structured workflow process definition</td>
<td>217</td>
<td></td>
</tr>
<tr>
<td>workflow process definition (WPD), synchronizing, see synchronizing workflow process definition</td>
<td>217</td>
<td></td>
</tr>
<tr>
<td>Workflow Process Definition Language (WPDL) 26</td>
<td></td>
<td></td>
</tr>
<tr>
<td>workflow process, case-based 14</td>
<td></td>
<td></td>
</tr>
<tr>
<td>workflow process, inter-organizational, see inter-organizational workflow process</td>
<td>26</td>
<td></td>
</tr>
<tr>
<td>workflow schema, see workflow process definition (WPD)</td>
<td>26</td>
<td></td>
</tr>
<tr>
<td>workflow standards</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Business Process Execution Language for Web Services (BPEL4WS), see Business Process Execution Language for Web Services (BPEL4WS)</td>
<td>26</td>
<td></td>
</tr>
<tr>
<td>Business Process Modeling Language (BPML), see Business Process Modeling Language (BPML)</td>
<td>26</td>
<td></td>
</tr>
<tr>
<td>Web Service Choreography Interface (WSCI), see Web Service Choreography Interface (WSCI)</td>
<td>26</td>
<td></td>
</tr>
<tr>
<td>workflow process definition (WPD), see workflow process definition (WPD)</td>
<td>26</td>
<td></td>
</tr>
<tr>
<td>XML Process Definition Language (XPDL), see XML Process Definition Language (XPDL)</td>
<td>26</td>
<td></td>
</tr>
<tr>
<td>workflow system (WF-system) 47</td>
<td></td>
<td></td>
</tr>
<tr>
<td>workflow system (WF-system), short-circuited, see short-circuited WF-system</td>
<td>26</td>
<td></td>
</tr>
<tr>
<td>workflow task structure 34</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Staffware 34</td>
<td></td>
<td></td>
</tr>
<tr>
<td>workflow, see business process</td>
<td></td>
<td></td>
</tr>
<tr>
<td>WPDL, see Workflow Process Definition Language (WPDL)</td>
<td>26</td>
<td></td>
</tr>
<tr>
<td>WSCI, see Web Service Choreography Interface (WSCI)</td>
<td>26</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XML document 144</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
XML document, consistent, see consistent XML document
XML Process Definition Language (XPDL) 26
XOR-join 15, 102, 193, 194, 205, 210, 213, 239, 246
see also AND-join, mismatch, OR-join
XOR-split 15, 101, 102, 210, 213, 241, 246
see also AND-split, mismatch, OR-split
XPDL, see XML Process Definition Language (XPDL)
XRL any_sequence 146, 158
XRL choice 146, 158
XRL condition 148, 164
XRL event 152, 153, 176
XRL event handler 152
XRL instance 33
see also XRL route
XRL mapping 154
XRL parallel_no_sync 146, 151, 159, 176
XRL parallel_part_sync 147, 151, 160, 176
XRL parallel_part_sync_cancel 147, 151, 162, 176
XRL parallel_sync 146, 159
XRL process definition 144
instance-based 143
no dead task 173
option to complete 173
proper completion 168
reduction 175
soundness 174
verification 167
XRL process definition, Original, see Original XRL process definition
XRL process definition, Revised, see Revised XRL process definition
XRL route 144, 154
see also XRL instance
XRL route element 144
XRL routing element 144
XRL routing element, complex 146
XRL routing element, simple 144
XRL sequence 146, 157
XRL task 144, 152, 153, 156, 174, 176
XRL terminate 144, 153, 155, 174, 176
XRL terminate handler 153
XRL wait_all 147, 152, 153, 162, 174, 176
XRL wait_any 147, 152, 153, 163, 174, 176
XRL while_do 148, 163
XRL, control passing in 144, 150
XRL, see eXchangeable Routing Language (XRL)
XSLT, see eXtensible Stylesheet Language Transformations (XSLT)

Y
yellow marking 86, 90
SUMMARY

The workflow process definition (WPD) of a workflow management system (WFMS) is an important concept in this thesis. If we compare a WFMS to a conveyor belt system, then a WPD can be compared to a physical layout of the conveyor belts. Thus, both a WPD and a physical layout determine how items are moved around in the corresponding system.

However, a WPD can be far more complex than a layout of a conveyor belt system, because a conveyor belt system is forced to adhere to physical constraints, whereas a WFMS is not: A conveyor belt system is tangible and moves tangible work items around, whereas a WFMS, being a piece of software, is intangible and moves intangible information items (cases) around. For example, in a WFMS, cases can be copied with ease, which enables the parallel processing of multiple tasks on one case.

Unfortunately, today’s WFMSs have almost no support for the verification of a WPD, even though there is a clear need for such a verification support. Because a WPD can be very complex, it may contain (very subtle) errors. If an erroneous WPD is put into production, then it might take a while before somebody realizes that there is something wrong, as the entire process is hidden in the WFMS. Only anticipated errors can be detected by the WFMS itself, the remaining errors can only be detected after having requested some reports from the WFMS and only if these reports contain sufficient information to detect the errors.

This thesis presents the WPD verification tool Woflan and its supporting concepts. Woflan maps a WPD onto a workflow net (WF-net,
which is a Petri net with some additional requirements) and can verify, before the WPD is taken into production, the soundness property and four inheritance relations for the resulting WF-net. Note that the mappings used by Woflan should be of high quality, as the results obtained for the WF-net should be transferable to the originating WPD. This thesis presents such high-quality mappings for a number of commercial WFMSs (IBM MQSeries Workflow, Staffware, COSA Workflow), for a commercial BPR tool (Protos), and for a research workflow prototype (XRL).

The soundness property is a minimal property any WPD should satisfy, and consists of the following requirements: (i) whatever happens, every case can be completed, (ii) after a case has been completed, no dangling references are left behind to that case, and (iii) every part of the WPD is necessary for some case. Woflan exploits an existing relation between soundness and the (in Petri-net world well-known) boundedness and liveness properties for diagnosis purposes. Based on this relation, this thesis introduces a diagnosis process for the soundness property, and a concept of behavioral error sequences. The diagnosis process guides the designer of an unsound WPD towards correcting this WPD, using diagnostic information like the behavioral error sequences. These sequences pinpoint decisions in the WPD that, when taken into production, lead to a violation of the soundness property. This thesis presents a number of case studies that have put several mappings, the diagnosis process, and the behavioral error sequences to the test. From these case studies, we conclude that Woflan successfully guides the designer of an unsound WPD towards correcting that WPD.

The four inheritance relations can be used to guarantee that two WPDs behave in a similar way: If two WPDs share such a relation, then cases can be transferred between the WPDs without problems. Woflan implements algorithms to compute these four relations. However, a straightforward exhaustive search algorithm for computing the most complex relation, the life-cycle inheritance relation, can lead to excessive processing times. This thesis presents a backtracking algorithm that aims to alleviate this problem of excessive processing times for computing the life-cycle inheritance relation. This thesis also presents a number of case studies that have put both algorithms to the test. From these case studies, we conclude that the backtracking algorithm effectively reduces excessive processing times. From one of these case studies, we also concluded that the life-cycle inheritance relation can be very subtle and, therefore, hard to check for humans (even for experts). Therefore, we conclude that we need a tool like Woflan to check this inheritance relation.
Een belangrijk begrip in dit proefschrift is de workflow process definition (WPD) van een workflow management system (WFMS). Als we een WFMS vergelijken met een lopende band systeem, dan kunnen we een WPD vergelijken met een opstelling van de lopende banden in het systeem. Immers, net als een opstelling van lopende banden bepaalt ook een WPD naar welke werkplek een werkstuk vervolgens naar toe gaat.

Een WPD kan echter veel ingewikkelder zijn dan een lopende band systeem, omdat een WPD zich niets hoeft aan te trekken van enige fysieke beperkingen. Immers, in tegenstelling tot een (tastbaar) lopende band systeem, dat tastbare werkstukken verplaatst, is een WFMS een systeem dat ontastbaar is (het is software) dat ontastbare informatiestukken verplaatst. Hierdoor is het bijvoorbeeld mogelijk dat in een WFMS op meerdere plaatsen tegelijk aan een casus wordt gewerkt.

Helaas bieden hedendaagse WFMSs nauwelijks technieken om WPDs te verifiëren, ondanks het feit dat er een duidelijke behoefte aan lijkt te zijn. Immers, omdat een WPD erg ingewikkeld kan zijn, kan het ook (erg subtiele) fouten bevatten. Als een foutieve WPD in productie wordt genomen, dan kan het even duren voordat iemand door heeft dat er iets mis is, omdat het geheel verborgen zit in het WFMS. Het WFMS kan zelf alleen bekende en dus verwachte fouten detecteren; voor het opsporen van onverwachte fouten is men afhankelijk van rapportages uit het WFMS en men moet maar hopen dat de fouten te achterhalen zijn uit deze rapportages.
Dit proefschrift presenteert het WPD verificatietool Woflan, tezamen met de concepten waarop het is gebaseerd. Woflan zet een WPD om naar een workflow net (WF-net, een Petri net met enkele beperkingen) en kan het resulterende WF-net verifiëren op de soundness eigenschap en op vier inheritance relaties. Natuurlijk moeten de omzettingen van dien aard zijn dat de resultaten die behaald worden voor het WF-net van toepassing zijn op de oorspronkelijke WPD. Dit proefschrift geeft zulke omzettingen voor een aantal commerciële WFMSs (IBM MQSeries, Staffware, MQSeries), voor een BPR tool (Protos), en voor een research prototype (XRL).

De soundness eigenschap is een minimale voorwaarde voor een willekeurige WPD, en bestaat uit de volgende drie eisen: (i) elke casus moet beëindigd kunnen ongeacht wat er gebeurt, (ii) een beëindigde casus laat geen loze verwijzingen achter in het WFMS, en (iii) elk deel van de WPD is nodig voor enige casus. Voor het verifiëren van soundness maakt Woflan gebruik van de bestaande relatie tussen soundness en de (in de Petri net wereld bekende) boundedness en liveness eigenschappen. Gebaseerd op deze relatie presenteert dit proefschrift een diagnose proces voor soundness en zogenaamde behavioral error sequences. Het diagnose proces helpt de ontwerper van een unsound WPD bij het opsporen van fouten, bijvoorbeeld door de ontwerper de behavioral error sequences te tonen. Een behavioral error sequence geeft de exacte lokatie aan van een beslissing in de WPD waarna de soundness eigenschap niet meer te redden is. Dit proefschrift geeft een aantal case studies die een aantal omzettingen, het diagnose proces, en de behavioral error sequences hebben getest. Uit deze case studies blijkt dat Woflan de ontwerper van een unsound WPD helpt bij het verbeteren van die WPD.

De vier inheritance relations garanderen, indien aanwezig, dat twee WPDs zich zodanig identiek gedragen dat casussen zonder problemen kunnen worden uitgewisseld tussen beide WPDs. Met Woflan is het mogelijk om deze vier relaties te berekenen. Echter, de meest complexe van de vier, life-cycle inheritance, kan leiden tot buitensporige berekeningstijden indien een standaard exhaustive search algoritme wordt gebruikt. Dit proefschrift geeft een backtracking algoritme dat deze buitensporige berekeningstijden moet voorkomen. Daarnaast geeft dit proefschrift een aantal case studies, waaruit blijkt dat dit backtracking algoritme inderdaad deze buitensporige berekeningstijden weet te voorkomen. Uit een van deze case studies hebben we ook geconcludeerd dat de life-cycle inheritance relatie erg subtiel kan zijn en daardoor erg moeilijk te controleren voor mensen (zelfs als het experts zijn). Hieruit concluderen we dat voor het bepalen van deze relatie een tool als Woflan onmisbaar is.
Eric Verbeek was born on October 16, 1965 in Eindhoven, the Netherlands, where he also grew up. From 1984 until 1988 he went to secondary school at the Van Maerlantlyceum in Eindhoven.

After he finished his secondary education, he started his studies in Computing Science (Technische Informatica) at the Technische Universiteit Eindhoven, the Netherlands. In May 1988, he graduated at this university under the supervision of Prof. Dr. K.M. van Hee.

From June 1988 to October 1988, he worked for five months as an engineer in the group where he graduated, which was more or less a natural extension of his graduation project. At the end of October 1988 he joined the military service for fourteen months.

In March 1990 he started working as a scientific engineer at the department of Mathematics and Computing Science of the Technische Universiteit Eindhoven, where he worked for several years on the ExSpect project under the supervision of Dr. L.J.A.M. Somers. During some of these years, he also held a part-time position at Bakkenist Management Consultants. As from 1996, he started working on Woflan under the supervision of Dr. ir. W.M.P. van der Aalst. In March 2000, he moved from the department of Mathematics and Computing Science to the department of Technology Management of the same university.

At the moment, Eric Verbeek is still working as a scientific engineer at the department of Technology Management. His main research area is process verification, in particular verification of workflow processes using Petri nets. He can be reached at h.m.w.verbeek@tm.tue.nl.